Pillole di Large Scale and Multi-Structured Database

Questi appunti sono stati scritti con l'intento di ripassare/studiare il programma di *Large Scale and Multi-Structured Database* durante la preparazione dell'esame. Non costituiscono dunque una dispensa completa sul corso ma piuttosto un manuale d'uso per affrontare la preparazione dell'esame con serenità e senza farsi prendere dal panico. L'ordine con il quale sono stati trattati gli argomenti non rispecchia l'ordine con cui il professore li ha presentati in classe ma piuttosto ho scritto questi appunti seguendo un flusso di coscienza.
In bocca al lupo colleghi, con affetto <3

Federico Casu

Key-Value databases

Il nostro percorso alla scoperta dei NoSQL databases incomincia dai *key-value database*. I key-value sono la forma di database più semplice che esista: di fatto la struttura logica del database si sviluppa sull'idea di *array*!

Come ben sappiamo, un ***array*** è una struttura dati che può essere descritta come una lista ordinata di coppie <indice, valore>. Ormai siamo già da un po' nel business dei computer ed un concetto elementare come l'array non ci dovrebbe spaventare in alcun modo. Gli array, nella forma di liste ordinate di coppie <indice, valore>, hanno alcune limitazioni:

1. L'**indice** può essere solo uno scalare intero.
2. Tutti i **valori** dell'array devono essere di uno ed un solo tipo.

Le due limitazioni elencate sopra non vengono prese e buttate nel cestino dagli ***array associativi***:

- Un *array associativo* generalizza l'idea di lista ordinata presentata in prima istanza dagli array *normali*.
- Negli *array associativi* l'indice può essere di qualsiasi tipo: intero, carattere, stringa, ecc. ecc.
- I valori memorizzati nell'array associativo non devono rispettare alcun vincolo di tipo; quindi, contemporaneamente, possono coesistere valori di tipi diversi nel medesimo array.

Ed ecco che si arriva ai ***key – value*** databases: quest'ultimi sono dei databases la cui struttura logica di memorizzazione è l'array associativo! Ovviamente, affinché possano essere considerati database, i dati sono memorizzati persistentemente.

L'unico requisito necessario è il seguente:

> Ogni valore deve avere un **identificatore univoco** nella forma della chiave. In particolare, una chiave deve essere univoca all'interno del *namespace* nel quale è definita.

Quali sono le features di un *key – value database*?

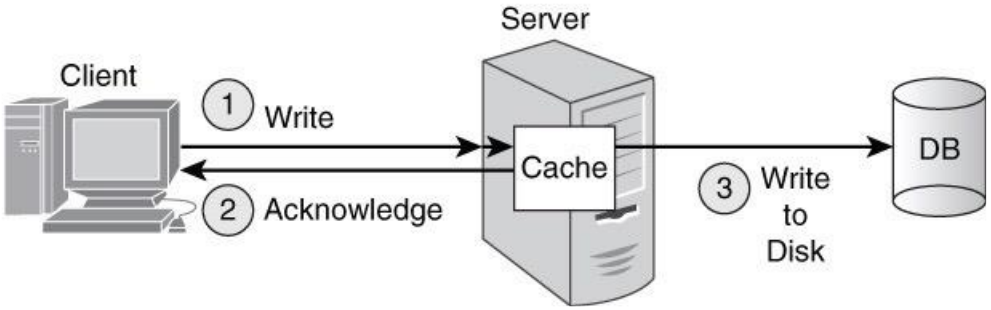
1. ***Semplicità***
2. ***Velocità***
3. ***Scalabilità***

Gli sviluppatori scelgono i key – value databases quando non occorre avere una struttura complessa (come quella offerta dai database relazionali) ma, invece, si predilige la facilità di accesso ai dati e il risparmio di spazio.

Analizziamo le tre proprietà sopra elencate:

1. Semplicità – In molte occasioni non è necessario avere le features offerte dai database relazionali (i.e. *join*). In questi casi i key – value offrono una struttura logica semplice ed efficace per la memorizzazione e lettura/scrittura dei record (l'accesso è semplice e veloce grazie all'indice).
   La semplicità dei key – value è dovuta anche al fatto che non necessitano di uno schema dei dati definito a priori (ovvero il contrario di quanto è necessario in un database relazionale). Facciamo un esempio: se l'applicazione che fa uso di un key – value database necessita di memorizzare nuovi attributi, non è necessario modificare il database visto che non esiste alcun schema da rispettare; è sufficiente modificare il codice dell'applicazione per gestire i nuovi attributi aggiunti.
   Inoltre, i key – value databases non necessitano che i valori memorizzati siano omogeni nel "tempo" e nello "spazio": lo sviluppatore può assegnare un intero ad una certa chiave e poi in futuro può assegnare una stringa alla stessa chiave senza alcun problema.
2. Velocità – Ovviamente, come tutti i databases che si rispettano, i dati devono essere memorizzati persistentemente. Questo implica che le letture/scritture dei dati siano

soggette a ritardi dovuti alla natura del dispositivo di memorizzazione di massa. Qui entra in gioco il concetto di ***cache***. Le letture da memoria RAM sono indubbiamente più veloci rispetti alle letture su dispositivi di archiviazione di massa.



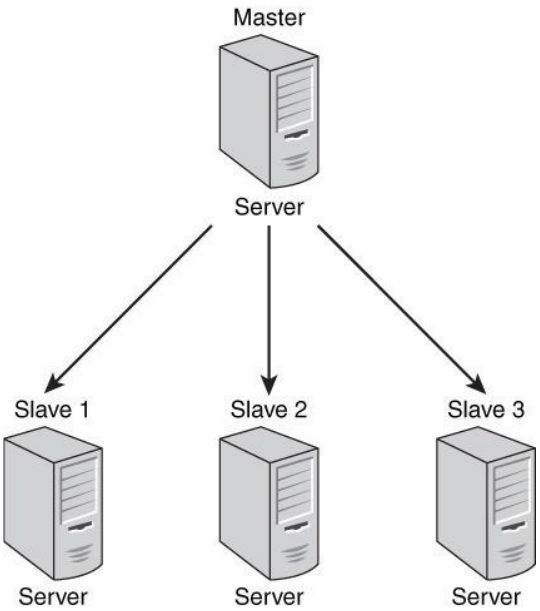
Purtroppo, la memoria RAM non è infinita: quando lo spazio a disposizione finisce è necessario spostare alcuni record dalla memoria RAM alla memoria persistente. Esistono diversi algoritmi che selezionano quali saranno i record oggetto di swap; uno tra tutti è ***LRU*** (Least Recent Used): questo algoritmo individua i record che sono stati letti/scritti meno di recente tra tutti.

3. Scalabilità – La ***scalabilità*** è la capacità di un sistema ad aggiungere/rimuovere servers da un cluster con l'intenzione di accomodare il carico del sistema. I key – value databases frequentemente utilizzano due soluzioni per scalare: ***master – slave replication*** e ***masterless replication***.

Scaling with master – slave replication

Questa è la soluzione più classica in commercio ed è anche quella più semplice ed efficace da un punto di vista di consistenza dei dati.

- Il ***master*** accetta tutte le richieste di *scrittura* e le richieste di *lettura*.
- Gli ***slaves*** accettano (e rispondono) solo alle richieste di *lettura*.
- Il master è l'unico server che gestisce le scritture: questa architettura non presenta il problema dell' inconsistenza dei dati perché le scritture sono gestite da un'unica entità che si occupa, nell'immediato futuro, di propagare le modifiche su tutti gli slaves.
- Questo tipo di architettura è consigliata nel caso l'applicazione in questione sia **read – heavy**.



Vediamo quali sono i PRO e i CONTRO della *master – slaves replication:*

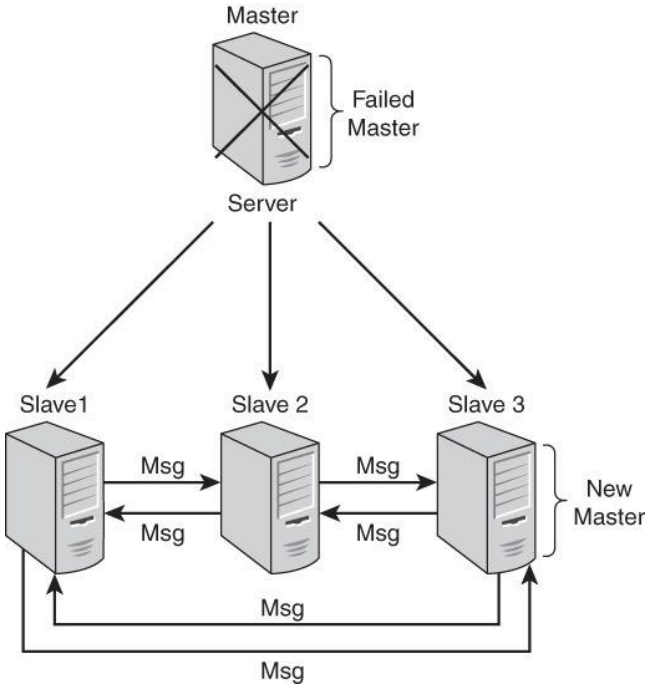
PRO:

1. L'architettura è molto semplice: solo il master comunica con gli altri server (non c'è bisogno di collegare gli slaves tra loro).
2. Non c'è alcun bisogno di coordinare le operazione di aggiornamento: il master (entità centrale) gestisce da solo queste operazione. Non ci sono conflitti tra aggiornamenti visto che nessun altro server, oltre al master, accetta richieste di scritture.

CONTRO:

1. *Single point of failure*: se master cade, l'applicazione perde tutte le funzionalità legate all'aggiornamento dei dati. Le funzionalità legate alla lettura dei dati non vengono perse perché gli slaves sono in grado di rispondere a tali richieste.

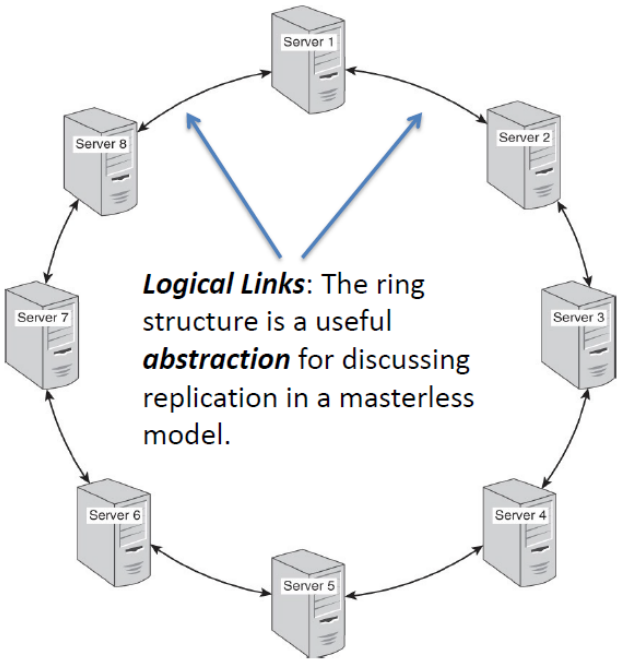
Possiamo risolvere il problema del *single point of failure* introducendo un meccanismo di elezione del master nel caso il master attuale non sia raggiungibile.

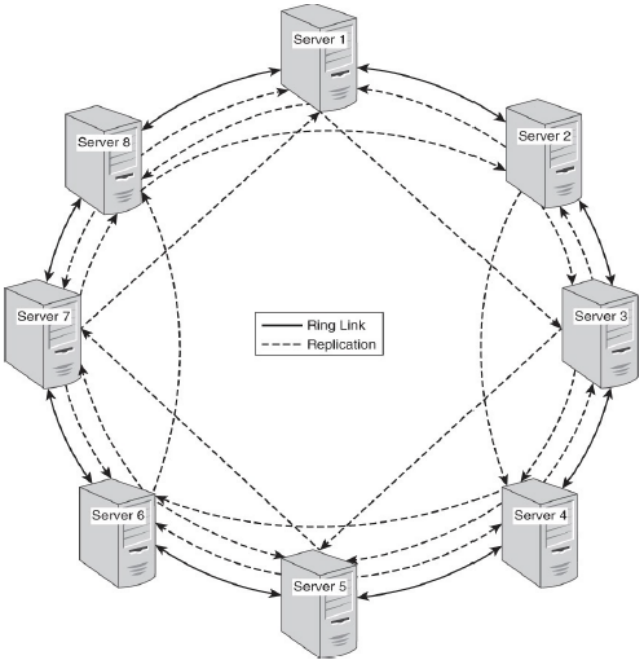


In questo caso, tutti i server presenti nell'architettura si mandano in modo randomico un heartbeat e sono così in grado di detectare se il master per qualche ragione va down e iniziare la procedura di rielezione.

Scaling with masterless replication

Introduciamo l'architettura *masterless* tramite uno scenario nel quale la *master – slaves* replication potrebbe rivelarsi la scelta peggiore possibile: the website which sells ticket of concerts.





- A differenza dell'architettura *master – slaves*, in una *masterless replication* esistono più server che possiedono la master copy dei dati.
- I server hanno il compito di aiutare i propri server vicini. Questo in cosa si traduce? Un esempio di implementazione di questa politica potrebbe essere la seguente: ogni qual

volta un server riceve gestisce una scrittura propaga quest'ultima ai suoi *due* server vicini e ad un server che si trova *due link* avanti a lui (in senso circolare).

Keys: more than meaningless identifiers

Nello scenario dei database relazionali *è bene* utilizzare delle chiavi primarie prive di significato. Vediamo un esempio che ci convincerà subito: supponiamo di avere un' e-commerce e supponiamo che si abbia una tabella nel quale vengono memorizzate tutte le informazioni dei clienti (nome, cognome, e-mail, indirizzo, metodo di pagamento, ecc. ecc.). Supponiamo che la chiave primaria che identifica i clienti sia composta dalla concatenazione del nome, cognome e codice postale. Se uno dei clienti cambiasse residenza, e quindi l'indirizzo di spedizione necessariamente deve essere aggiornato, allora è necessario non solo aggiornare la chiave primaria (visto che parte di essa è stata aggiornata con l'aggiornamento dell'indirizzo di spedizione) ma anche tutte le chiavi esterne che si riferisce alla chiave primaria dei clienti (quest'ultima operazione è necessaria per rispettare il vincolo di integrità).
Ecco perché, nel contesto dei database relazionali, è bene specificare chiavi primarie *meaningless*.

La situazione cambia quando si parla di key – value databases: in questo caso, visto che l'informazione può essere acceduta solo tramite chiave, è bene che le chiavi abbiano un significato. Supponiamo di avere, come prima, un' e-commerce e supponiamo di gestire ordini e clienti tramite key – value database. Supponiamo di salvare i prodotti di un ordine utilizzando come chiave un codice univoco:

order[67482] = {"Apple Iphone 14", "Cover Spigen Iphone 14"}

L'unico modo che abbiamo per accedere a quell'ordine è conoscere il codice che lo identifica. Ma poniamoci un altro problema: a quale cliente appartiene l'ordine 67482? Per risolvere entrambi i problemi potremo pensare di definire due ulteriori *namespace*. In particolare:

user[78451862] = "Federico Casu"
listOfOrders[78451862] = {67482, 61422, 57457}

Cosa abbiamo appena fatto? Una stronzata: stiamo ricalcando il modello relazione su un key – value database ☹.

> Possiamo definire **chiavi meaningful** che includono informazioni riguardo il ***nome dell'entità***, l'***identificatore dell'entità*** e l'***attributo rappresentato***.

La regola generale per definire una chiave è la seguente:

EntityName:EntityID:EntityAttribute

From keys to values

Visto che la struttura delle chiavi può essere di qualsiasi tipo e di qualsiasi forma, dobbiamo necessariamente trovare un modo per accedere ai valori in funzione della chiave data. Ecco che ci vengono in soccorso le *hash function*:

> Le **hash function** sono utilizzate per ottenere l' "indirizzo" al quale è memorizzato il valore associato ad una chiave. Le hash function ritornano un valore numerico a partire da una stringa (in questo caso la chiave) data in input.

Attenzione: esiste il problema della *collisione*!

Le *hash function* vengono utilizzate non solo per l'indirizzamento dei record ma anche per fare ***load balancing***!
Vediamo come:

1. Supponiamo di avere un'architettura masterless (per la precisione i server sono disposti secondo una *topologia ad anello*). Ogni server è identificato con un ID compreso tra [0, N].
2. Il load balancing è implementato in funzione dell'hash della chiave del record che deve essere scritto:
    a. Si calcola l'hash della chiave.
    b. Il record viene scritto sul server avente ID = modulo(*hash(key), N*).

Grazie a questo metodo è possibile risolvere un classico problema (che riguarda tutti i SQL/NoSQL in caso di *replication*).

Problem – To avoid selling tickets to the same seat, at the same venue, in the same city, on the same night to more than one person.
Solution – We can specify a key that identifies univocally the seat. If we use a load balancing policy based on hashing the key, anyone that will try to purchase that same seat on the same day would generate the same key, that will be transformed by the hashing + modulo function in the same Server Address → There is no chance for another server to sell that seat!

Operation allowed in key – value databases

Le uniche operazioni consentite dai key – value databases sono tre:

- Recuperare un dato tramite chiave. (GET)
- Settare un nuovo dato tramite chiave. (SET)
- Eliminare un dato esistente tramite chiave. (DELETE)

Si noti che non esiste un linguaggio per esprimere query più complesse. Se si necessitano di query più elaborate è necessaria implementarle direttamente lato applicazione (si, c'è del codice da scrivere).

Tips and tricks: keys

Come detto precedentemente, è bene che le chiavi siano *meaningful* visto che la chiave costituisce l'unico strumento utile per poter accedere al valore a cui si è interessati. In generale, le chiavi sono delle stringhe che seguono un pattern (quest'ultimo definito a priori dallo sviluppatore). Detto questo, è doveroso far presente che chiavi molto lunghe sono "pesanti" da memorizzare. Quest'ultimo costituisce un aspetto da non sottovalutare visto che i key – value databases sono spesso ***memory intensive***. D'altro canto, definire delle chiavi troppo corte può rappresentare un problema dal punto di vista delle collisioni (ricorda l'indirizzamento viene fatto sulla base dell'hash della chiave).

Tips and tricks: values

Dal punto di vista dei valori non abbiamo quasi nessuna limitazione. Un valore è un oggetto, ovvero un insieme di bytes associati ad un chiave. Di conseguenza un oggetto può essere di qualsiasi tipo: *intero*, *float*, *stringa* o oggetti complessi come *immagini*, file *JSON*, ecc. ecc.
Le uniche limitazioni imposte sui valori potrebbero riguardare la dimensione di questi ultimi. In ogni caso, le limitazioni dipendono dal framework utilizzato dunque è sempre bene consultare la documentazione fornita con il framework.

Tips and tricks: namespaces

Un namespace è un insieme di coppie <chiave, valore> il cui unico requisito è l'univocità della chiave all'interno di quel namespace. Di fatto, nella stessa istanza di key – value database, possono coesistere due record aventi la medesima chiave a patto che appartengono a due namespace distinti.

Data partitioning

Con *data partitioning* intendiamo quel processo in cui, preso in considerazione l'insieme D di tutti i dati, D viene suddiviso in *n* sottoinsiemi (disgiunti) ed ognuno di questi viene gestito da un server (istanza di database) del cluster.
Questa pratica è utile nel caso in cui si voglia fare *load balancing*: le letture/scritture che un server deve gestire riguardano solamente il sottoinsieme di dati che gestisce; come conseguenza, il numero di letture/scritture che il server si troverà a gestire diminuisce rispetto allo scenario nel quale il server gestisce l'intero insieme D.

Qual è l'obiettivo del *data partitioning*? ***Distribuire equamente i dati tra tutti i server del cluster***. Per raggiungere questo obiettivo è importante notare che, a partire dalla partition key, la scelta del criterio sul quale partizionare l'insieme dei dati non è banale. Introduciamo un esempio illuminante:

- Supponiamo di voler distribuire equamente le coppie <chiave, valore> di un key – value database tra *n* server di un cluster. Supponiamo inoltre di voler usare la chiave come partition key e di voler partizionare l'insieme dei dati in funzione della prima lettera della chiave.
- Notiamo subito che la strategia di partitioning scelta non ci condurrà mai ad avere una suddivisione equa del carico perché le chiavi sono costruite secondo un certo pattern e di conseguenza hanno una struttura molto simile. Nel caso degenere, se il nostro insieme di dati consistesse in un solo namespace e le chiavi fossero costruite a partire da un pattern del tipo *namespace:entityName:entityID:entityAttribute*, allora verrebbe popolato solamente il server associato alla prima lettera del *namespace*.

La soluzione a questo problema, ancora una volta, è semplice ed elegante: le *hash function*! Infatti, gli algoritmi di *hashing* sono tali da distribuire equamente gli output sull'intero codominio.

Come viene implementato il *data partitioning*? Semplice: tramite *hash function* e *partition key*. Facciamo un esempio pratico:

1. Supponiamo di aver un cluster composto da *n* server. Ogni server gestisce una partizione dei dati D.
2. Selezioniamo una *partition key*, ovvero una chiave sulla base della quale selezioniamo il server sul quale memorizzare il dato.
3. Per selezionare il server (quindi la partizione al quale appartiene il dato) spesso si utilizzano le *hash functions*.

In Key – Value databases, clusters tend to be ***loosely coupled***. This means that the server ***are independent*** and complete many functions on their own with ***minimal coordination*** with other servers in the cluster.
Each server is ***responsible*** for the operations on ***its own partitions*** and routinely ***send messages*** to each other to indicate they are still functioning. When a node ***fails***, the other nodes in the cluster can respond by ***taking over the work*** of that node.

Consistent hashing

Poniamoci il seguente problema: cosa succede se aggiungiamo/rimuoviamo un nodo dal nostro cluster su cui abbiamo fatto data partitioning?
Purtroppo, la risposta non ci piacerà: dobbiamo modificare la *hashing function* e riallocare i dati tra i server rimanenti/nuovi (a seconda se si sta rimuovendo un nodo o aggiungendone uno nuovo).
La soluzione a questo spiacevole inconveniente è data dal ***consistent hashing***. L'idea sul quale si basa il *consistent hashing* è di sfruttare la topologia ad anello. Supponiamo di avere un cluster composto da *n* server. Il server sul quale allocare l'oggetto in questione viene scelto secondo questa politica:

O_i sarà memorizzato sul nodo N_j se *hash(*N_j*)* è il ***successivo*** di *hash(*O_i*),* in *senso circolare*. (*hash(*N_j*)* e *hash(*O_i*)* deve appartenere allo stesso spazio di hashing).

Starting point:

employee_id	first_name	last_name	address
1	John	Doe	New York
2	Benjamin	Button	Chicago
3	Mycroft	Holmes	London

FOREIGN KEY

payment_id	employee_id	amount	date
1	1	50,000	01/12/2017
2	1	20,000	01/13/2017
3	2	75,000	01/14/2017
4	3	40,000	01/15/2017
5	3	20,000	01/17/2017
6	3	25,000	01/18/2017

Final point:

```
employee:1:first_name = "John"
employee:1:last_name = "Doe"
employee:1:address = "New York"


employee:2:first_name = "Benjamin
employee:2:last_name = "Button"
employee:2:address = "Chicago"


employee:3:first_name = "Mycroft"
employee:3:last_name = "Holmes"
employee:3:address = "London"


payment:1:1:amount = "50000"
payment:1:1:date = "01/12/2017"


payment:2:1:amount = "20000"
payment:2:1:date = "01/13/2017"


payment:3:2:amount = "75000"
payment:3:2:date = "01/14/2017"


payment:4:3:amount = "40000"
payment:4:3:date = "01/15/2017"


payment:5:3:amount = "20000"
payment:5:3:date = "01/17/2017"


payment:6:3:amount = "25000"
payment:6:3:date = "01/18/2017"
```

Design tips

Visto che l'unico strumento utile per accedere ai dati è la chiave, è bene definire un pattern (e seguirlo) per costruire tutte le chiavi necessarie. Un esempio potrebbe essere il seguente:

entityName:entityID:entityAttribute

Grazie a questo pattern è possibile definire metodi *getter/setter* in maniera semplice e leggibile:

```
define getCustomerName(string id)
{
        string key = "cust" + ":" id + ":" + "custName"
        return Customer[key]
}

define setCustomerName(string id, string name)
{
        string key = "cust" + ":" id + ":" + "custName"
        Customer[key] = name
}
```

Si noti che grazie al fatto che le chiavi rispettano il pattern *entityName:entityID:entityAttribute*, è possibile scrivere solo due metodi per fare get/set dei valori (quindi generalizzare le operazioni get/set):

```
define getCustomerAttribute(string id, string attribute)
{
        string key = "cust" + ":" id + ":" + attribute
        return Customer[key]
}

define setCustomerAttribute(string id, string attribute, string value)
{
        string key = "cust" + ":" id + ":" + attribute
        Customer[key] = value
}
```

Come possiamo gestire le query che necessitano la gestione di valori *range-based*? Ancora una volta dobbiamo ringraziare i noi del passato che hanno pensato di definire le chiavi secondo un pattern prestabilito. Facciamo un esempio: recuperare l'incasso totale proveniente dagli ordini evasi in una certa data.

```
define getTotalSalesByDate(string date)
{
        int total = 0
        int id = 0

        while(id <= threshold)
                string key = "order" + ":" id + ":" + "date" + ":" + date + ":" + "amount"
                if(exists(key))
                        total += Order[key]
                id++

        return total
}
```

Attenzione: non è oro tutto ciò che luccica! Supponiamo di avere un sito di e-commerce e supponiamo di gestire il sito mediante un key – value database. Una delle operazioni più comuni su questa tipologia di sito è sicuramente il checkout. Per poter recuperare le informazioni necessarie a svolgere correttamente il checkout dell'ordine potrebbero essere necessarie le seguenti informazioni:

1. Nome, cognome e indirizzo dell'utente.
2. Dati relativi al metodo di pagamento.
3. Sconti accumulati dal cliente.

Vediamo quali sono le operazioni necessarie per recuperare le informazioni elencate sopra:

```
define checkout(string customer_id, string order_id)
{
        string first_name = getCustomerAttribute(customer_id, "firstName")
        string last_name = getCustomerAttribute(customer_id, "lastName")
        string address = getCustomerAttribute(customer_id, "address")
        string payment_method = getCustomerAttribute(customer_id, "paymentMethod")
        string discounts = getCustomerAttribute(customer_id, "discounts")

        return new Checkout(first_name, last_name, address,
                                        payment_method, discounts, order_id)
}
```

È semplice notare che sono necessari 5 accessi al DB! Sicuramente questo approccio è poco efficiente. Un modo per ottimizzare gli accessi al DB è definire valori complessi! Visto che i key – value non impongono (generalmente) limitazioni sui tipi dei valori, possiamo pensare di raggruppare i valori frequentemente acceduti insieme in un valore complesso come un oggetto JSON.

Document databases

Gli sviluppatori spesso si rivolgono ai ***document databases*** quando hanno bisogno della flessibilità dei NoSQL databases ma devono gestire strutture di dati più complesse di quelle supportate dai *key – value* databases. Come i key – value databases e diversamente dai database relazionali, i document databases non richiedono la definizione di uno schema.
I document DB, tuttavia, hanno alcune caratteristiche simili a quelli relazionali. Ad esempio, è possibile interrogare e filtrare raccolte di documenti tanto quanto faresti in database relazionale. Naturalmente, la sintassi, o struttura, delle query è diverso tra database SQL e NoSQL, ma le funzionalità sono comparabili.

Ma alla fine della fiera, che cosa sono i document DB? I document sono dei database ***non relazionali*** che memorizzano i dati sotto forma di documenti, tipicamente documenti XML o JSON. Vien da sé che ci dobbiamo studiare sia i documenti XML sia i documenti JSON.

XML documents

XML sta per ***eXtensible Markup Language***. XML non è altro che un linguaggi di markup che, tramite ***tags***, specifica la struttura del documento oltre che il contenuto. Grazie proprio al fatto che XML definisce sia la struttura sia il contenuto, mediante i documenti XML è possibile rappresentare praticamente qualsiasi tipo di informazione.

Vantaggi di XML

1. I file XML sono text (Unicode) based.
2. Un documento XML può essere visualizzato diversamente a seconda del software che si utilizza per la visualizzazione.
3. I documenti XML sono modulari. Parti di un documento possono essere riutilizzati in contesti diversi senza dover riscrivere la struttura di quel modulo.

Nell'ecosistema XML, oltre ai documenti di tipo XML, esiste una tipologia speciale di documento: ***XML schema***. Grazie a quest'ultimi è possibile definire quali contenuti devono essere presenti all'interno di un documento XML.

Svantaggi di XML

1. XML tags sono verbosi e ripetitivi: lo spazio necessario per la memorizzazione aumenta
2. In generale, i documenti XML sono uno spreco di spazio e sono *computationally expensive* to parse.

JSON documents

JSON sta per ***JavaScript Object Notation***.

Esempio di un oggetto JSON:

```
{
  "firstName": "Jonathan",
  "lastName": "Freeman",
  "loginCount": 4,
  "isWriter": true,
  "worksWith": ["Spantree Technology Group", "InfoWorld"],
  "pets": [
    {
      "name": "Lilly",
      "type": "Raccoon"
    }
  ]
}
```

Esempio di un oggetti innestati:

```
{
  "sammy" : {
    "username"  : "SammyShark",
    "location"  : "Indian Ocean",
    "online"    : true,
    "followers" : 987
  },
  "jesse" : {
    "username"  : "JesseOctopus",
    "location"  : "Pacific Ocean",
    "online"    : false,
    "followers" : 432
  },
  "drew" : {
    "username"  : "DrewSquid",
    "location"  : "Atlantic Ocean",
    "online"    : false,
    "followers" : 321
  },
  "jamie" : {
    "username"  : "JamieMantisShrimp",
    "location"  : "Pacific Ocean",
    "online"    : true,
    "followers" : 654
  }
}
```

Esempio di array innestati:

```
{
  "first_name" : "Sammy",
  "last_name" : "Shark",
  "location" : "Ocean",
  "websites" : [
    {
      "description" : "work",
      "URL" : "https://www.digitalocean.com/"
    },
    {
      "desciption" : "tutorials",
      "URL" : "https://www.digitalocean.com/community/tutorials"
    }
  ],
  "social_media" : [
    {
      "description" : "twitter",
      "link" : "https://twitter.com/digitalocean"
    },
    {
      "description" : "facebook",
      "link" : "https://www.facebook.com/DigitalOceanCloudHosting"
    },
    {
      "description" : "github",
      "link" : "https://github.com/digitalocean"
    }
  ]
}
```

JSON vs XML

Similarità:

- Entrambi sono *human readable*.
- Entrambi hanno una sintassi semplice.
- Entrambi sono basati su una *struttura gerarchica*.
- Entrambi sono *language independent*.
- Entrambi sono supportati da APIs da utilizzare con i linguaggi di programmazione.

Differenze:

- La sintassi è differente.
- JSON è meno verboso rispetto XML.
- In JSON non possono essere presenti keyword di Javascript.

Ora entriamo più nel dettaglio nella terminologia dei document DB.

Document

Un ***documento*** è un insieme di coppie <chiave, valore> ordinate. Un coppia <chiave, valore> è una struttura dati che consiste di due parti chiamate, non a caso, la *chiave* e il *valore*. Non ci addentreremo nel dettaglio nel definire *chiave* e *valore* visto che sono due concetti già incontrati durante lo studio dei *key – value* DB.

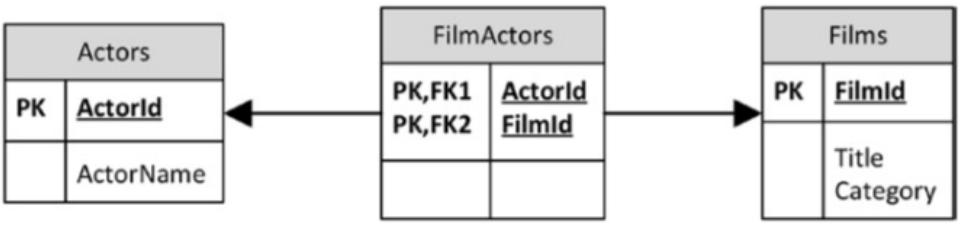
Possiamo dire che un documento, essendo un insieme, contiene una sola istanza di ogni coppia <chiave, valore>. Inoltre, come già visto nei key – value DB, il valore può essere “semplice” (quindi un intero, un numero con virgola, una stringa, un booleano, ecc. ecc.) oppure può essere complesso (quindi un array di valori, un array di documenti, ecc. ecc.). Ancora una volta, un valore è un *oggetto*: non abbiamo specificato alcun vincolo sul tipo del valore definendolo come un oggetto, quindi, può essere di qualsiasi tipo purché sia accettato dallo standard utilizzato (XML o JSON).

Collection

Una ***collection*** è un insieme di documenti. I documenti all'interno di una collezione sono di solito relativi alla stessa entità come dipendenti, prodotti, eventi registrati o clienti. È possibile archiviare documenti non correlati in una raccolta, ma ciò (ovviamente) non è consigliato.

Document embedding

Supponiamo di voler modellare il seguente scenario: vogliamo tenere traccia dei film in cui hanno recitato un insieme di attori. Se avessimo a disposizione un database relazionale allora lo schema (in terza forma normale) sarebbe il seguente:



Supponiamo ora di avere a disposizione un document DB. Come gestiamo il problema? Sostanzialmente abbiamo due metodi: *document embedding* e *document linking*. Analizziamo la prima soluzione:

> Fare ***document embedding*** consiste nell'innestare altri documenti all'interno di un campo di un documento con l'intenzione di rappresentare una relazione.

Riprendendo l'esempio da cui siamo partiti, possiamo rappresentare la relazione *molti a molti* tra l'entità *Attore* e l'entità *Film* in due modi equivalenti:

Film document

```
{
        "_id" : 83,
        "Title" : "CAMPUS REMEMBER",
        "Category" : "Action",
        "Actors" : [
                {"actor_id" : 811, "Name" : "Matthew Johanssons"},
                {"actor_id" : 741, "Name" : "Reese Kilmer"},
                {"actor_id" : 482, "Name" : "Will Wilson"}
        ]
}
```

oppure,

Actor document

```
{
        "_id" : 811,
        "Name" : "Matthew Johanssons",
        "Films" : [
                {"film_id" : 83, "Title : "CAMPUS REMEMBER"},
                {"film_id" : 97, "Title : "PORTLAND"},
        ]
}
```

Entrambe le soluzioni sono corrette: la scelta ricadrà sulla soluzione più efficiente ed utile dipendentemente dalle letture/scritture effettuate dall'applicazione.

Attenzione: questa soluzione non è priva di difetti. Consideriamo la soluzione con document embedding sulla collection *Film*. Possiamo notare che le informazioni di un attore sono presenti sia nella collection *Film* sia nella collection *Actor* → presenza di duplicati! Ogni qual volta si deve aggiornare un documento presente nella collection *Actor* è necessario aggiornare tutti i documenti della collection in cui è presente parte dell'informazione aggiornata dell'attore in questione.

^-^ <3 → By Caterina Bruchi

Document linking

Esiste un altro modo per rappresentare le relazioni senza dover fare *document embedding*: il *document linking*.
Consideriamo il seguente esempio che mappa le relazioni tra editore e libro:

Library collection:

```
{
    name: "O'Reilly Media",
    founded: 1980,
    location: "CA",
    books: [123456789, 234567890, ...]
}
```

Come possiamo vedere, il campo "*books*" consiste in un array di id, ciascuno di essi rappresenta un libro nella relativa collection. Utilizzando questo metodo non abbiamo fatto altro che ricalcare lo schema di un database relazione ☹.

Anche se questa strategia non è la più naturale da utilizzare quando si fa il design di un document DB, talvolta può essere la soluzione migliore se la cardinalità della relazione rappresentata è molto grande. In questo caso il *document linking* è la soluzione che offre il bilancio migliore tra *performance* e *manutenibilità*.

Data modelling: one to many relationship

Supponiamo di avere l'entità *customer* e supponiamo che ogni customer dell'applicazione possa avere più di un indirizzo di spedizione. Di fatto dobbiamo rappresentare una relazione *uno a molti* (lato uno: customer, lato molti: indirizzo).

Customer document:

```
{
   customer_id: 76123,
   name: 'Acme Data Modeling Services',
   person_or_business: 'business',
   address : [
                    { street: '276 North Amber St',
                      city: 'Vancouver',
                      state: 'WA',
                      zip: 99076} ,
                    { street: '89 Morton St',
                      city: 'Salem',
                      state: 'NH',
                      zip: 01097}
                   ]
   }
```

Una relazione ***one to many*** può essere tradotta rappresentando il lato molti della relazione come array di documenti associati ad un campo dell'entità lato uno (*document embedding*).

Data modelling: many to many relationship

Supponiamo di avere il seguente scenario:

- Uno studente universitario può essere iscritto ad uno o più corsi.
- Un corso universitario può avere uno o più studenti iscritti.

Come possiamo rappresentare una relazione molti a molti nei document DB? La soluzione è semplice: *document linking* (*document embedding* se la dimensione dei documenti è ragionevolmente piccola).

Course collection:

```
{
  { courseID: 'C1667',
     title: 'Introduction to Anthropology',
     instructor: 'Dr. Margret Austin',
     credits: 3,
     enrolledStudents: ['S1837', 'S3737', 'S9825' …
       'S1847'] },
  { courseID: 'C2873',
     title: 'Algorithms and Data Structures',
     instructor: 'Dr. Susan Johnson',
     credits: 3,
     enrolledStudents: ['S1837','S3737', 'S4321', 'S9825'
       … 'S1847'] },
  { courseID: C3876,
     title: 'Macroeconomics',
     instructor: 'Dr. James Schulen',
     credits: 3,
     enrolledStudents: ['S1837', 'S4321', 'S1470', 'S9825'
       … 'S1847'] },
```

Student collection:

```
  {
   {studentID:'S1837',
     name: 'Brian Nelson',
     gradYear: 2018,
     courses: ['C1667', C2873,'C3876']},
   {studentID: 'S3737',
     name: 'Yolanda Deltor',
          gradYear: 2017,
          courses: [ 'C1667','C2873']},
      …
  }
```

Ovviamente si deve prestare attenzione nel momento in cui si compie un aggiornamento che coinvolge almeno una delle due collection: al contrario di quanto accade nei database relazionali, non esiste alcun ***vincolo d'integrità*** da rispettare e di conseguenza un aggiornamento potrebbe portare a situazione di inconsistenza tra le due collection.

Data modelling: an example

Scenario: Trucks in a company fleet have to transmit location, fuel consumption and other metrics every three minutes to a fleet management data base (***one*** – ***to*** – ***many*** relationship between a truck and the transmitted details).

Quale può essere una possibile soluzione? Supponiamo di creare una nuova collection, chiamiamola *Detail*, e supponiamo che ogni documento della collection abbia la seguente struttura:

Document Detail
```
{
    “truck_id” : 574125,
    “truck_driver” : “Arthur Shelby”,
    “timestamp” : “2023:02:14-14:57”,
    “fuel_consumption” : “14.2 MPG”,
    “other_info” : “”
}
```

Questo soluzione, seppur sia corretta, non è molto efficiente dal punto di vista dello spazio. Se prevediamo una giornata lavorativa di 10 ore ed una trasmissione ogni 3 minuti, a fine giornata avremo 200 documenti nuovi per ogni camion. Magari il numero non ci spaventa ma il problema più grande è dovuto al fatto che ogni documento riporta delle informazioni “duplicate” ad ogni trasmissione (vedi truck_id, truck_driver).

Ma allora come possiamo risolvere questo problema? Semplice: utilizziamo la tecnica del *document embedding!* In particolare inseriamo un array di documenti all’interno di ogni documento che identifica il trasporto.

Document Transport
```
{
    “truck_id” : 574125,
    “truck_driver” : “Arthur Shelby”,
    “log_book” : [
                    {“timestamp” : “2023:02:14-14:57”, “fuel_consumption” : “14.2 MPG”},
                    {“timestamp” : “2023:02:14-15:00”, “fuel_consumption” : “14.7 MPG”},
                    {“timestamp” : “2023:02:14-15:03”, “fuel_consumption” : “13.9 MPG”}
                ]

}
```

Quando viene creato un documento, il DBMS alloca una certa quantità di spazio per il documento. Se il documento *cresce* fino a terminare lo spazio allocato, il DBMS deve “trasferirlo” in un'altra posizione. Inoltre, il DBMS deve *liberare* lo spazio precedentemente assegnato.

➔ Le operazioni necessarie per allocare un nuovo documento in un embedded array potrebbero influire negativamente sulle performance dell’intero sistema.

Una soluzione per evitare di allocare nuovo spazio e liberare lo spazio precedentemente occupato da documenti di grandi dimensioni è allocare, quindi ***a priori***, una quantità sufficiente di spazio nel momento in cui il documento viene creato.

Indexing document DB

Quando abbiamo a che fare con databases di grandi dimensioni e, soprattutto, dobbiamo rispettare dei non–functional requirements che impongono dei requisiti sulle perfomance dell’applicazione (fast reads) è bene utilizzare gli ***indici***.

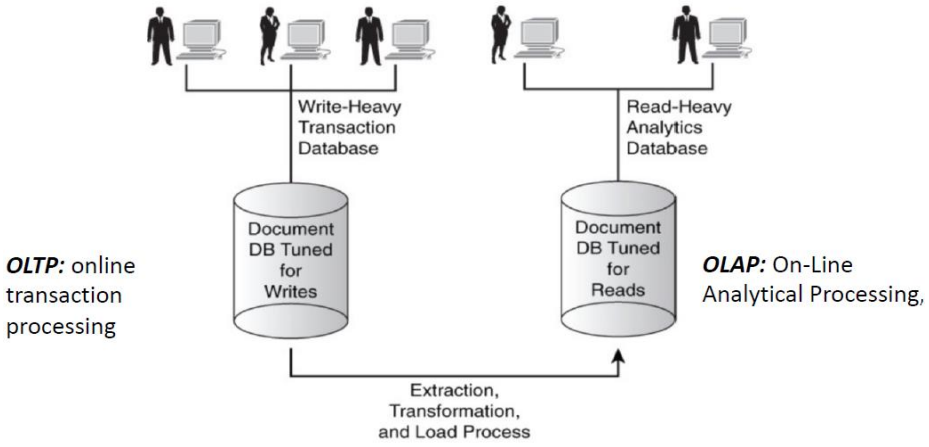
Un ***indice*** è una struttura dati che memorizza una porzione dei dati con una struttura che ottimizza la ricerca e la lettura dei dati. In particolare, un indice è una struttura dati che memorizza i valori di un campo (sottoinsieme dei campi) del documento in modo ordinato. La memorizzazione ordinata ottimizza tutte le operazioni di ricerca e lettura.

Se la nostra applicazione è ***read heavy***, ovvero il workload è per la maggiore composto da operazioni di lettura, allora l'utilizzo di indici porta un miglioramento delle prestazioni in termini di diminuzione del tempo di risposta.

D'altro canto, se la nostra applicazione è ***write heavy***, ovvero il workload è per la maggiore composta da operazioni di scrittura, l'utilizzo di indici porta un peggioramento delle prestazioni. Questo è dovuto al fatto che, ad ogni operazione, l'indice deve essere aggiornato: se aggiungiamo un nuovo documento allora anche l'indice deve essere aggiornato.

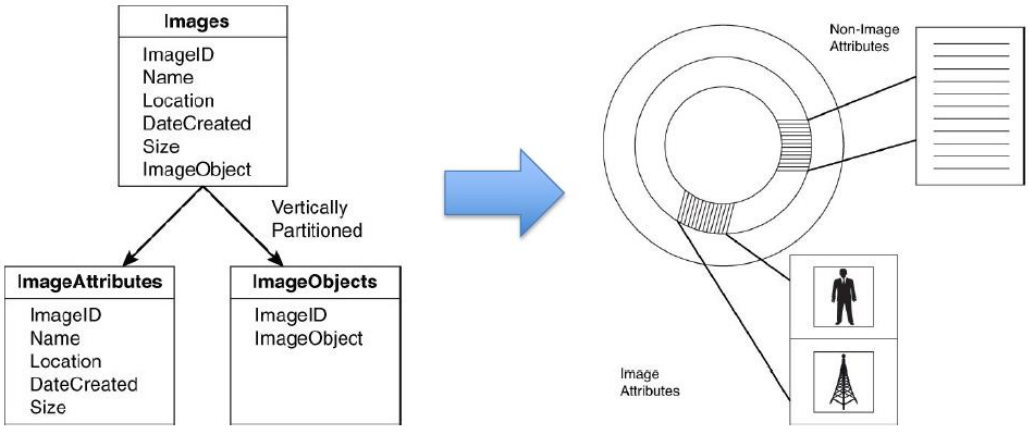
Ancora una volta, non esiste la soluzione perfetta: la scelta è guidata dalla natura dell'applicazione. Se la nostra applicazione è *read heavy* allora è sensato inserire un indice per ogni campo (sottoinsieme di campi) del documento soggetto ad un grosso numero di operazioni di lettura; ovviamente la nostra applicazione sarà *veloce* nel rispondere ad una richiesta di lettura ma sarà *lenta* a riacquisire il controllo quando compie un operazione di scrittura.

Una tipica soluzione con il quale si possono ottenere sia *fast reads* sia *fast writes* è la seguente:



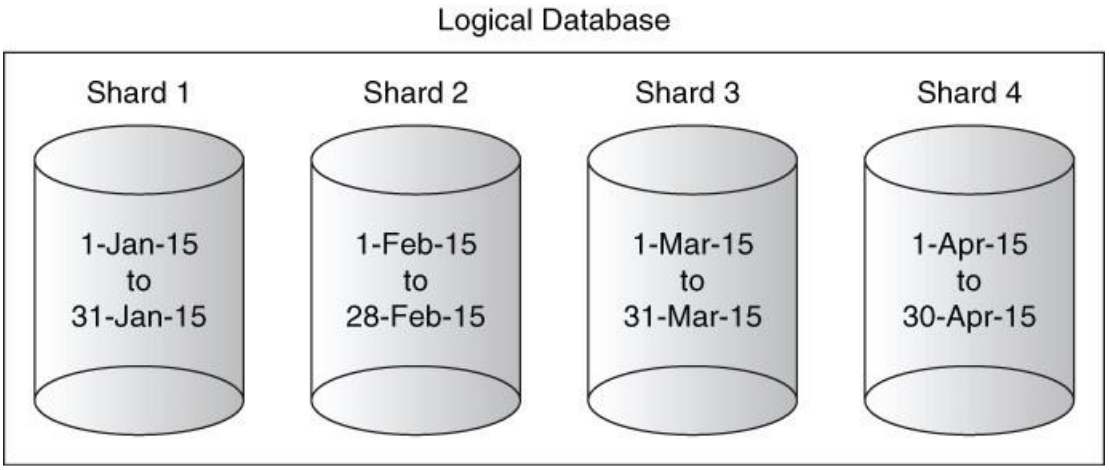
Vertical partitioning

Il ***vertical partitioning*** è una tecnica utilizzata per migliorare le prestazioni di un database relazionale. Il vertical consiste nel separare l'allocazione in memoria delle colonne di una tabella in più sottoinsiemi di colonne, ciascuno di essi allocato in una partizione di memoria dedicata.



Sharding

Lo ***sharding*** (*partizionamento orizzontale*) è il processo di divisione di una collection in *n* sottoinsiemi oppure, in un database relazionale, dividere una tabella per righe. Queste parti del database, note come ***shards*** (*frammenti*), sono archiviati su server separati. Un singolo shard può essere memorizzato su più server quando un database è configurato per le repliche. Indipendentemente dal fatto che i dati vengono replicati o meno, un server all'interno di un document DB cluster memorizzerà un solo shard.



Vantaggi dello sharding

- Meno costoso rispetto al *vertical scaling* (quest'ultimo consiste nell'aumentare le capacità di un singolo server aggiornando la CPU, aumentando la memoria RAM, ecc. ecc.).
- Insieme alle *repliche*, lo sharding assicura high availability e fast responses.
- Consente la gestione di alti workloads e consente la gestione di situazioni in cui si avverte un incremento di utenti che utilizzano il servizio.
- I dati possono essere distribuiti in maniera semplice su un numero variabile di server.

Come si fa sharding? Abbiamo bisogno di una ***shard key*** e di un ***algoritmo di partitioning***.

> ***Shard key***: una shard key è un sottoinsieme di attributi, quest'ultimi presenti in tutti i documenti della collection, sul quale viene deciso il partizionamento orizzontale dei documenti.

> ***Algoritmo di partitioning***: algoritmo che, dato in input la shard key, suddivide la collection in *n* partizioni disgiunte; quest'ultime costituiscono gli shard da distribuire sulle istanze di document DB attive all'interno del cluster.

Tipi di algoritmi di partitioning:

1. ***Range*** – Un partizionamento secondo range può essere fatto, ad esempio, su un campo che riporta la data di creazione. L'anno solare viene suddiviso in *n* intervalli (dove *n* è il numero di istanze di document DB attive nel cluster). Un documento viene assegnato al server *i* se la data di creazione appartiene a $[n_i, n_{i+1}]$.
2. ***Hashing*** – Il partizionamento viene eseguito facendo hashing della shard key. Il documento viene memorizzato nel server cui indice è dato da *i = modulo(hash(shard_key), n)*.
3. ***List*** – I documenti vengono partizionati sulla base del *tipo* ottenuto dal valore della *shard key*. Esempio: i documenti possono essere partizionati in funzione se rappresentano una prodotto tecnologico, un capo d'abbigliamento, ecc. ecc.

Can we store documents regarding different “types” of entity in the same collection?

La domanda è semplice per cui anche la risposta lo è: SI. Attenzione: seppur in linea teorica i document DB non impongono alcuni vincolo riguardante il *tipo* dell’entità modellata da un documento, dobbiamo prestare attenzione a quando si fa una scelta di questo tipo. Vediamo il perché:

- *Mischiare* documenti relativi ad entità diverse può condurre ad avere documenti che modellano entità diverse nella stessa partizione del disco (quello nella quale vengono memorizzati i documenti della collection).
- Questo porta ad inefficienze. Prendiamo lo scenario nel quale l’applicazione filtra i documenti per tipo e, la maggior parte delle volte, richiede i documenti relativo ad uno specifico tipo. Ovviamente si capisce che mischiare documenti di due o più tipi, quando se ne richiede per la maggior parte delle volte solo uno, non è la soluzione migliore.

Potrebbe essere utile usare questo approccio quando i documenti di due (o più) tipi distinti vengono richiesti insieme.

Column family databases

Prima di descrivere cosa sono i *column DB* facciamo un passo indietro e introduciamo il momento storico da cui deriva l'esigenza di avere dei column DB.

OLTP vs OLAP

- OLTP, ovvero OnLine Transaction Processing – Vengono definite OLTP tutte quelle architetture software orientare alla gestione di operazioni CRUD.
- OLAP, ovvero OnLine Analitycs Processing – Vengono definite OLAP tutte quelle architetture software orientate alla gestione di analisi (veloce) di dati.

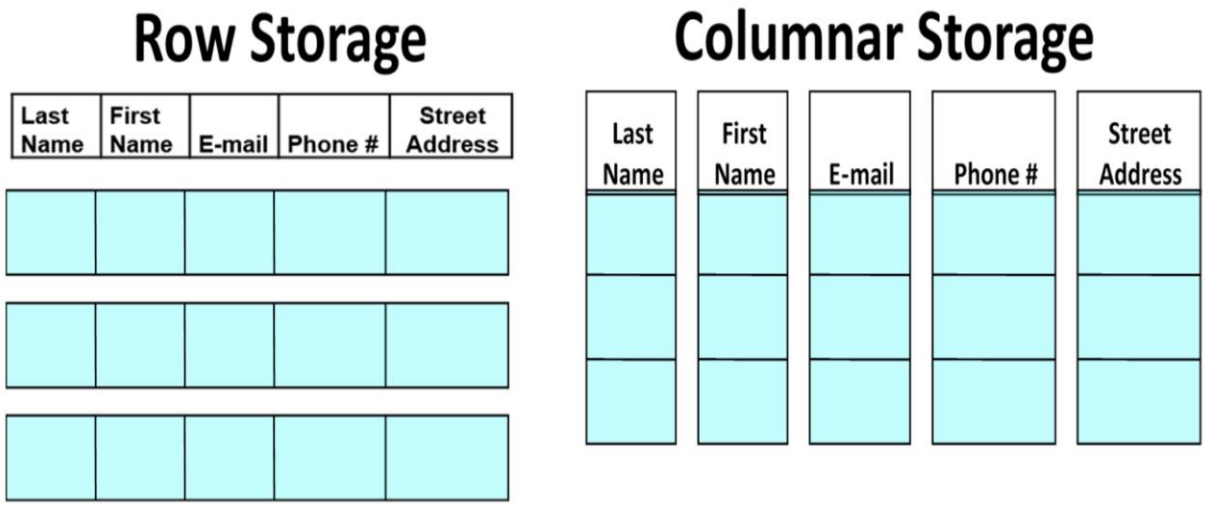
Perché questa distinzione? All'inizio dell'era digitale i file erano strutturati secondo una struttura fisica organizzata per ***rows***. I software detti OLTP sono nati con l'obiettivo di gestire le operazioni CRUD di file con una struttura fisica *row – oriented.*

Quando i ***business intelligence softwares*** iniziarono a prendere piede allora iniziò a diffondersi il bisogno di avere "riposte" in tempi brevi.

La riposta all'esigenza di avere ***faster response times*** è data dai *column family DBs*! Vediamo perché:

- Ragionando in termini di database relazioni, le analitycs vengono eseguite su un sottoinsieme delle colonne di una tabella ma coinvolgono tutti i valori di una certa colonna. Esempio: trovare il dipendente con il salario minimo. Per fare un analitycs di questo tipo sono necessarie solo due informazioni relative al dipendente (nome e salario). D'altro canto, per trovare il salario minimo si devono confrontare i salari di tutti i dipendenti (quindi si accede ad ogni row della tabella).
- Riprendendo l'esempio fatto sopra, e supponendo che ogni row della tabella sia allocata in un blocco fisico del disco, eseguire l'analitycs richiede l'accesso di *n* blocchi in memoria fisica, dove *n* è la cardinalità dell'insieme dei dipendenti.
- Ecco spiegato il motivo degli ***slow response times***: per eseguire un analitycs, seppur non troppo complessa, è necessario compiere un considerevole numero di accessi al disco.

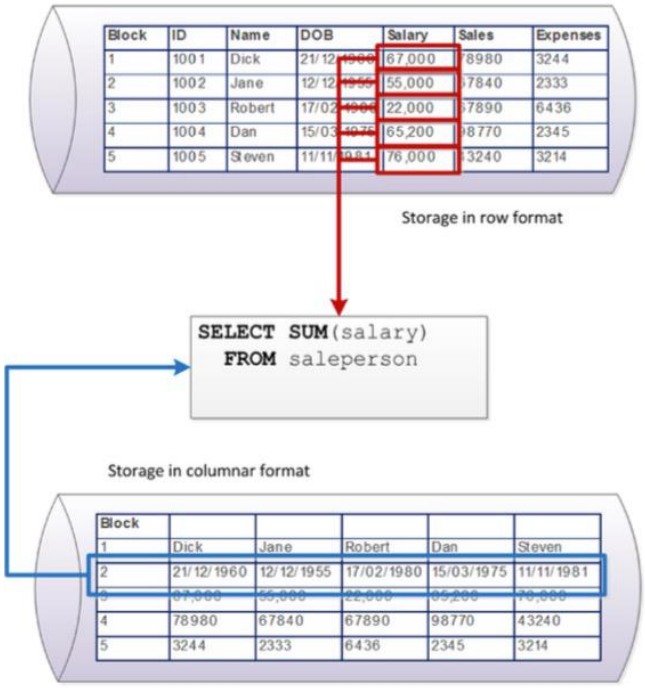
L'idea alla base dei column DB è semplice: raggruppare tutti i valori di un attributo in un unico blocco. Ciò si traduce nell'allocare una colonna (o un sottoinsieme di colonne) all'interno di un blocco.



Facciamo un esempio: supponiamo di voler calcolare il totale speso dall'azienda in salari dei dipendenti.

L'immagine sulla destra è auto esplicativa:

1. Se avessimo a disposizione un database organizzato per righe (banalmente un database relazionale), allora è necessario, in generale, accedere 5 volte alla memoria fisica per recuperare i salari dei dipendenti.
2. Invece, se il nostro database fosse organizzato per colonne (ed ogni colonna fosse allocata in un blocco del disco) allora risulta sufficiente accedere solo una volta al disco per recuperare tutti i salari dei dipendenti.



Un altro aspetto interessante offerto dai column family DBs è la ***facilità di risparmiare spazio***. Con i column DB è possibile raggiungere ***high compression ratio*** con poco effort computazionale. Facciamo un esempio: supponiamo di avere un attributo i cui valori sono interi; se memorizziamo quell'attributo con una colonna e memorizziamo i valori ordinati allora è possibile memorizzare solo il primo valore e memorizzare i successivi in funzione della differenza tra il valore corrente ed il precedente.

Delta store

Problema: l'architettura base dei column family DB non è grado di gestire efficientemente stream costanti di operazioni di modifica che, nel dominio dei database relazionali, riguardano l'inserimento/modifica di una riga della tabella.
Soluzione: il ***delta store*** è un'area del database mantenuta in memoria avente la caratteristica di non essere compressa e di essere ottimizzata nel gestire operazioni frequenti di modifica dei dati.
Ovviamente, questo tipo di soluzione, porta ad avere dei dati non aggiornati se le operazioni di merge tra delta store e column DB non vengono fatte frequentemente. Per eseguire le analitycs potrebbe essere necessario dover accedere al delta store per avere i dati aggiornati.

Projections

Problema: query complesse talvolta necessitano dati provenienti da un gruppo di colonne.
Soluzione: talvolta i column family DB adottano il meccanismo delle proiezioni. Una proiezione di fatto è un gruppo di colonne che vengono memorizzate insieme (quindi presumibilmente nello stesso blocco). Le colonne che compongono la proiezioni sono accomunate dal fatto che sono accedute molto spesso insieme.

BigTable

TODO

Keyspace

Un keyspace è l'analogo dei namespace per i key-value DB. All'interno di un keyspace sono contenute le colonne, le row keys e tutte le relative strutture dati.

Row Keys

Le ***row keys*** sono uno dei componenti che permettono di identificare *univocamente* i valori memorizzati in un column DB. Inoltre, analogamente alle *keys* nei key – value DB, le row keys sono utilizzate per partizionare, e ordinare, i dati. Ad esempio, in *Cassandra*, le row keys sono utilizzate da un modulo del DBMS, detto *partitioner*, per ordinare i dati e distribuirli casualmente tra i nodi del cluster.

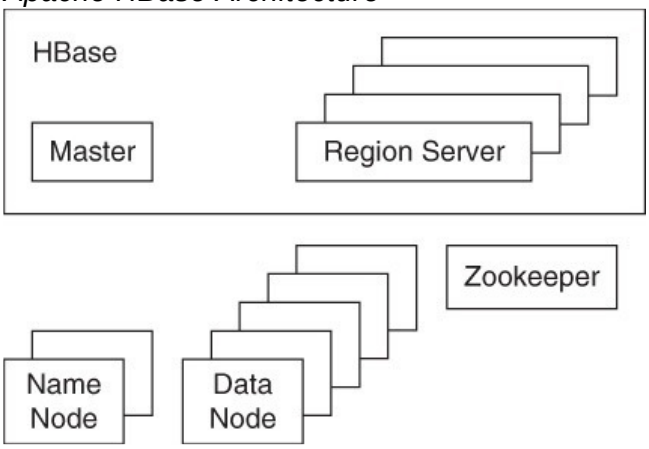
Columns

Le *columns*, insieme alle row keys e alla versione del dato (molto spesso rappresentato da un timestamp), sono un modo per identificare univocamente i dati. Inoltre, esistono le *column families*, ovvero gruppi di colonne frequentemente usate insieme. Le column families possono essere modificate dinamicamente, ovvero una colonna può essere aggiunta/rimossa a posteriori senza necessita di rispettare uno schema definito all'inzio.

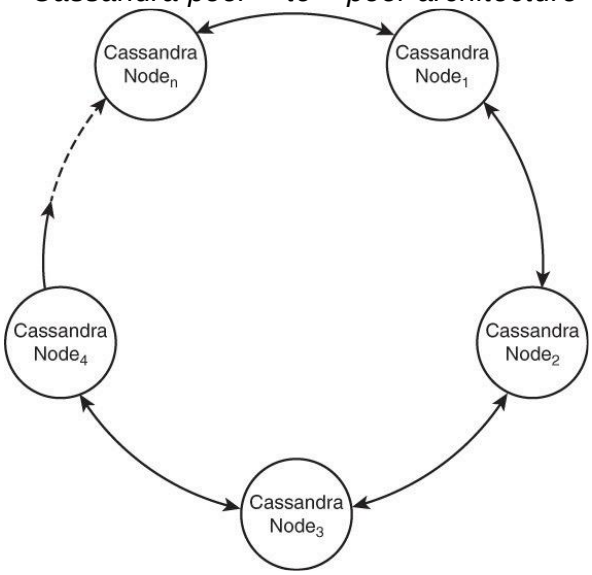
Column DB architectures

In commercio esistono due tipi di architetture:

Apache HBase Architecture



Cassandra peer – to – peer architecture



HBase – L'architettura HBase si basa sull'esistenza di diversi nodi che ricoprono ruoli diversi all'interno del cluster. In particolare, abbiamo I seguenti ruoli:

1. *Hmaster*: nodo centrale che assegna le regioni ai *region servers*.
2. *Region server*: nodo che accetta tutte le richieste (lettura/scrittura) provenienti dalla regione assegnata dal Hmaster. Si noti che i dati contenuti in un region server sono relativi alla partizione associata alla regione. Facciamo un esempio: se Netflix usasse il column DB messo a disposizione da Apache allora la regione potrebbe rappresentare una nazione e la partizione di dati mantenuta dai region servers conterrebbe il catalogo relativo alla nazione.
3. *Zookeeper*: è un tipo di nodo che si occupa di coordinare i nodi all'interno del Hadoop cluster.

Cassandra – Tipica architettura a ring dove tutti i nodi possono espletare tutte le operazioni (non esistono nodi di tipi diversi).
Vantaggi:

1. Semplicità
2. Nessun nodo costituisce un *single point of failures*.
3. Scala facilmente.
4. I server comunicano tra di loro senza aver bisogno di un'entità terza che coordini le operazioni.
5. Se un nodo viene rimosso dal cluster, i nodi che mantegono le repliche del nodo in questione possono rispondere alle richieste. Successivamente, quando il nodo viene reinserito, i nodi replica possono propagare le modifiche.

Commit logs

Per velocizzare le operazioni di scrittura molto spesso vengono utilizzati i *commit log*. Un commit log non è altro che un file *append – only* nel quale viene scritto un record per ogni richiesta di modifica che l'istanza DB riceve. Il controllo viene restituito all'applicazione appena la richiesta viene scritta sul commit log. Questo approccio permette di diminuire i tempi d'attesa in scrittura (scrivere un record nel commit log è molto più veloce rispetto a modificare il DB). Inoltre, visto che i commit log mantengono lo storico delle modifiche, è possibile recuperare lo stato del DB anche in caso di errori.

Bloom filters

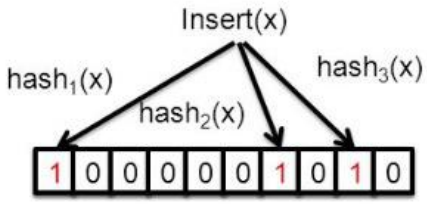
I *bloom filters* sono una struttura dati ***probabilistica***. L'obiettivo dei bloom filters è ridurre il numero di accessi durante un operazione di lettura. In particolare, l'obiettivo dei bloom filters è non accedere alle partizioni nel quale non è presente il dato oggetto di ricerca.

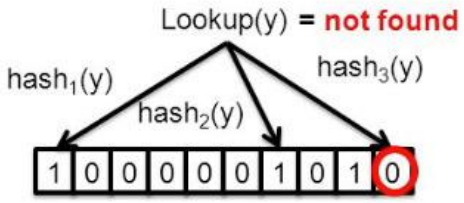
Un bloom filter viene usato quando un client performa una richiesta di lettura: il bloom filter risponde ***maybe*** se il dato è presente nel set su cui viene eseguito il test oppure ***no*** se il dato non appartiene al set.

Quando il test restituisce ***no*** allora il dato non è, con certezza, presente nel set su cui viene eseguito il test.

Descriviamo il funzionamento dei ***bloom filters*** con un esempio:

- Descriviamo le operazioni di scrittura con la funzione ***Insert****(d)*.
- Descriviamo le operazioni di lettura con la funzione ***Lookup****(d).*
- Supponiamo di avere una partizione. In questo caso un bloom filter può essere rappresentato da un array binario di *n* elementi. Inoltre, consideriamo un set di *k hash functions* (dove *k* << *n*).
- Inizialmente ogni elemento del bloom filter è inizializzato a zero. Ogni qual volta il nodo intercetta una richiesta di scrittura, Insert(d_0), entrano in gioco le funzioni di hashing. Viene calcolato l'hash del dato da inserire con ogni funzione (quindi si ottengono *k* hash). Come ultima operazione vengono settati a 1 tutti gli elementi dell'array il cui indice è stato restituito dalle *k* funzioni di hashing.
- Per quanto riguarda le richieste di lettura, Lookup(d_0), si calcolano *k* hash a partire dal dato oggetto di ricerca. I k hash ottenuti sono gli indici dell'array che devono essere testati: se *anche solo un* elemento dell'array è diverso da 1 allora il dato oggetto di ricerca è certamente non presente.



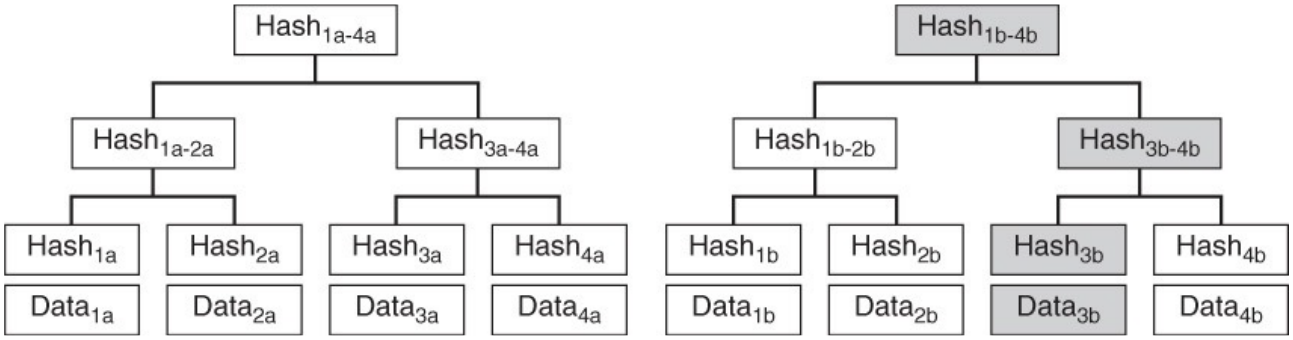


Un importante proprietà dei *bloom filters* è la seguente: il tempo necessario ad aggiungere un elemento / testare se l'elemento è presente è ***costante***. Inoltre, non è possibile rimuovere un elemento da un bloom filter.

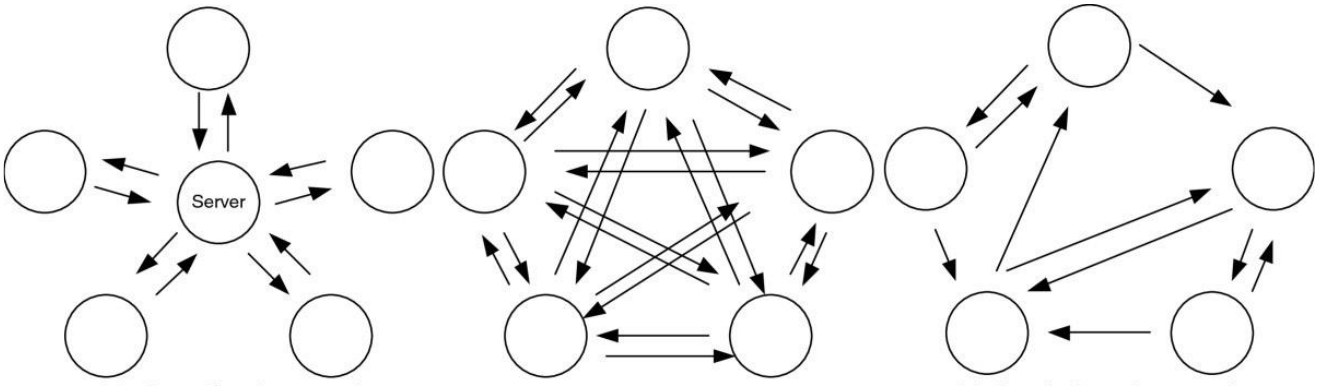
Anti Entropy

L'***anti entropy*** è il processo di verificare la presenza di differenza tra repliche. Come si può implementare un meccanismo di questo tipo? Attraverso *hash trees*:

- Le foglie dell'albero contengono l'hash di una certa partizione
- I nodi contengono l'hash di tutti i loro figli.



Communication protocols



(a) Centralized approach, which uses a centralized server for forwarding and mixing of multimedia streams

(b) Fully connected overlay approach, where all peers are in direct contact with one another

(c) Gossip-based approach, where peers operate in parallel, and each peer communicates with one or more randomly selected partner

Hinted Handoff

Le repliche ci assicurano ***high availability*** in lettura anche se alcuni nodi non sono raggiungibili. Purtroppo, *high availability in lettura* non implica *high availability in scrittura*. Lo ***hinted handoff*** è un meccanismo progettato per risolvere questo problema.

Se un'operazione di scrittura viene indirizzata a un nodo non disponibile, l'operazione può essere reindirizzata a un altro nodo, quest'ultimo può essere un' altra replica o un nodo designato per ricevere le richieste di scrittura quando il nodo di destinazione è inattivo.

Il nodo *sostitutivo* crea una struttura dati per archiviare le informazioni sull'operazione di scrittura e le informazioni necessarie a propagare l'operazione una volta che il nodo diventa nuovamente raggiungibile. Il meccanismo di hinted handoff controlla periodicamente lo stato del server di destinazione e invia l'operazione di scrittura quando il nodo è nuovamente disponibile.

L'archiviazione di una struttura di dati che memorizza le informazioni sull'operazione di scrittura non è la stessa cosa della scrittura su una replica.

Una volta che l'operazione di scrittura è stata propagata correttamente al nodo di destinazione, tale scrittura è considerata corretta ai fini della *consistency* e *replication*.

Column DB vs Relational DB

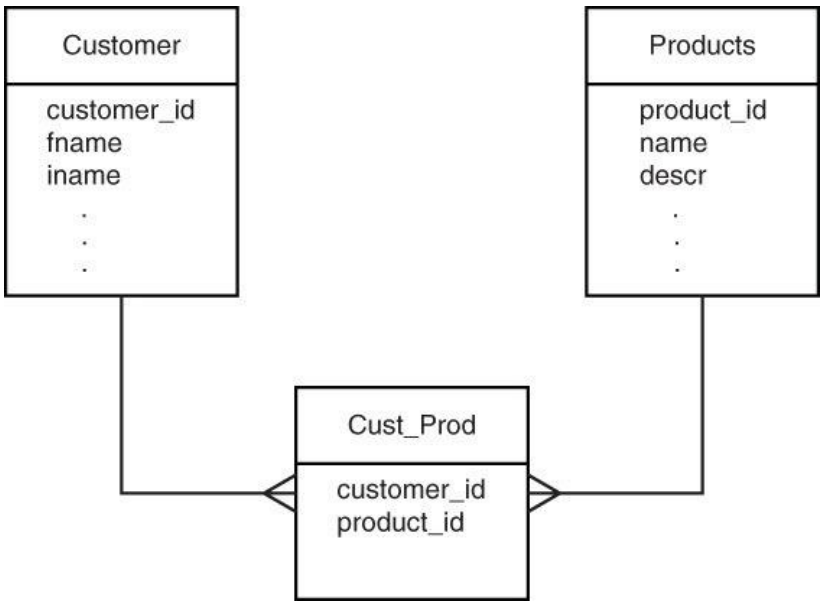
Quali sono le differenza tra column DBs e database relazionali?

1. I column DB sono implementati in modo ***sparso*** e attraverso ***mappe multidimensionali***.
2. Nei column DB non tutte le row hanno lo stesso numero di colonne (di fatto i column DB sono schemaless!).
3. Nei column DB le colonne possono essere aggiunte dinamicamente (anche questo è conseguenza del fatto che i column DB sono schemaless).
4. Nei column DB i *join* non sono utilizzati perché i dati sono de–normalizzati.

Street	City	State	Province	Zip	Postal Code	Country
178 Main St.	Boise	ID		83701		U.S.
89 Woodridge	Baltimore	MD		21218		U.S.
293 Archer St.	Ottawa		ON		K1A 2C5	Canada
8713 Alberta DR	Vancouver		BC		VSK 0AI	Canada

How to model many to many relationships using Column DB

Supponiamo di avere un sito che si occupa di vendere prodotti online e vogliamo tenere traccia dei prodotti comprati da un certo cliente. Di fatto, clienti e prodotti sono legati da una relazione molti a molti. Se utilizzassimo un database relazionale allora la relazione si tradurrebbe in tre tabelle: *customer, products* e *cust_prod* dove quest'ultima è stata introdotta per rappresentare la relazione molti a molti.



La relazione tra clienti e prodotti può essere rappresentata de–normalizzando i dati! In particolare, se siamo interessati a vedere tutti i prodotti comprati da un cliente allora potremo costruire una tabella in cui riportiamo come *row keys* gli id dei clienti mentre come column name

riportiamo gli id dei prodotti (si, possiamo memorizzare informazioni non solo nei column value ma anche nelle column key).

Customer

Row key	fname	lname	street	city	state
123	Jane	Smith	387 Main St	Boise	ID
287	Mark	Jones	192 Wellfleet Dr	Austin	TX
1987	Harsha	Badal	298 Commercial St	Provincetown	MA
2405	Senica	Washington	98 Morton Ave	Windsor	CT
3902	Marg	O'Malley	981 Circle Dr	Santa Fe	NM

Product

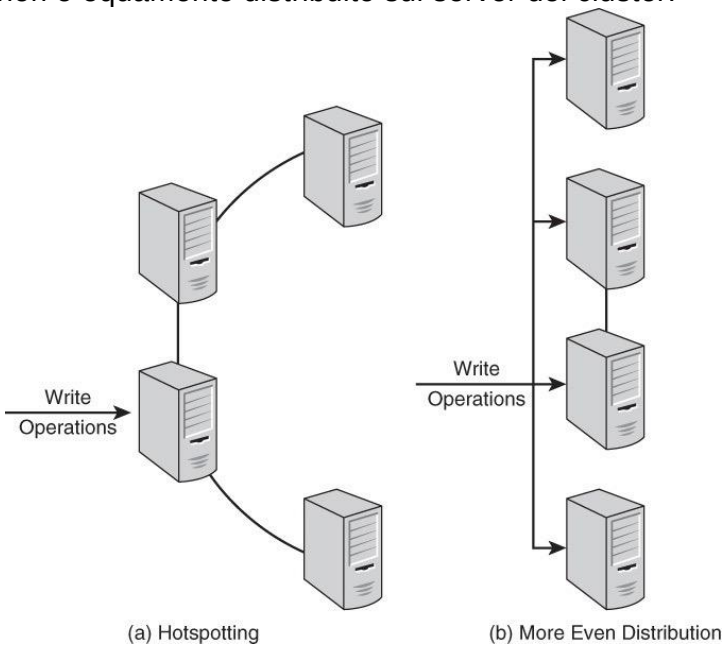
Row key	name	descr	qty_avail	category	
38383	Dell Latitude E6410	Laptop with ...	124	Computer	
48282	Apple iPhone	iPhone 6 with ...	345	Phone	
59595	Galaxy Tab S	Samsung tablet ...	743	Tablet	

Prod_by_Cust

Row key	38383	48282	59595		
123	Dell Latitude E6410	Apple iPhone			
287		Apple iPhone			
1987	Dell Latitude E6410				
2405			Galaxy Tab S		
3902		Apple iPhone			

Avoid hotspotting in row keys

Che cosa s'intende per hotspotting? Lo ***hotspotting*** accade quando il carico causato da richieste di scrittura non è equamente distribuito sui server del cluster.



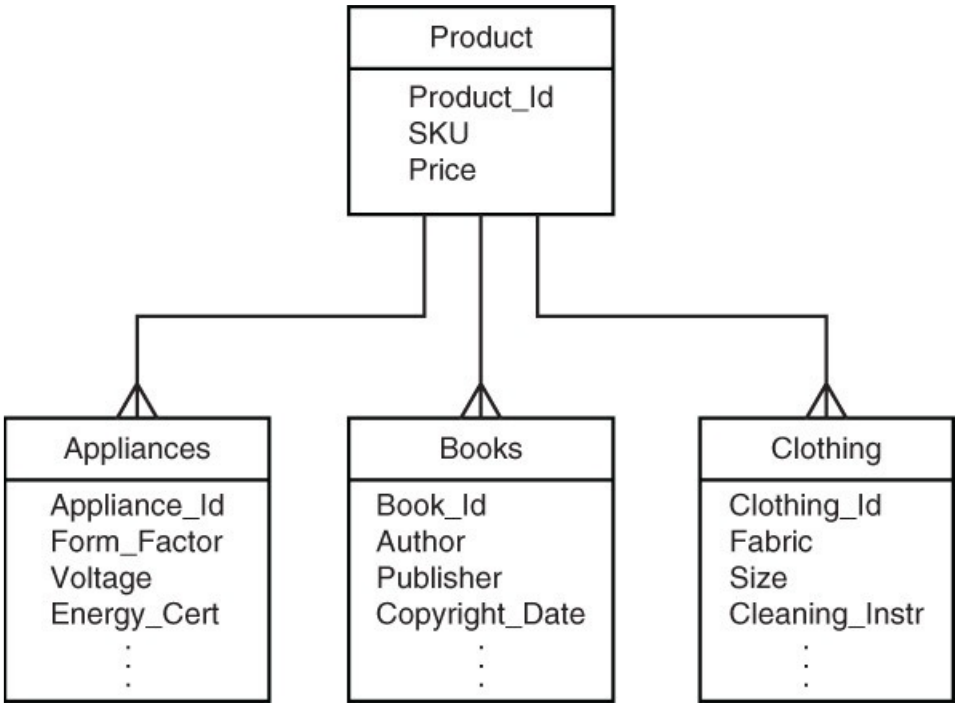
You can prevent hotspotting by hashing sequential values generated by other systems. Alternatively, you could add a random string as a prefix to the sequential value. This would eliminate the effects of the lexicographic order of the source file on the data load process.

Avoid complex data structures in column values

Perchè è bene poter utilizzare le features offerte dai DB (come indexes)! Inoltre, ricordiamo che possiamo modellare strutture dati complesse attraverso le column families!

Model an entity with a single row

Nei column family DB le operazioni di scrittura sono atomiche a livello di ***row***. Supponiamo di avere la seguente gerarchia:



Come è possibile modellare la gerarchia senza perdere l'atomicità della scrittura di una nuova istanza di prodotto? Possiamo sfruttare le ***column families***:

- Ricordandoci che le scritture sono *atomiche* a livello di ***row***, possiamo modellare la gerarchia con mediante ***una sola tabella e 3 column families***:
  - ***Product's attributes + Appliances' attributes***
  - ***Product's attributes + Books' attributes***
  - ***Product's attributes + Clothing's attributes***

Indexing: primary and secondary indexes

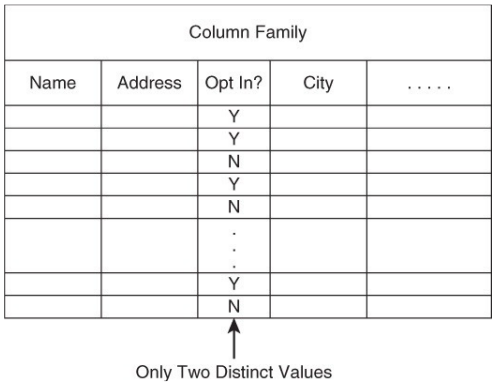
Come visto nei *document DB*, un ***indice*** è una struttura dati strutturata in modo tale da permettere la ricerca, ed il "recupero", di dati in maniera veloce ed efficacie dal punto di vista computazionale.

Nei column families DB esistono due tipologie di *indici*:

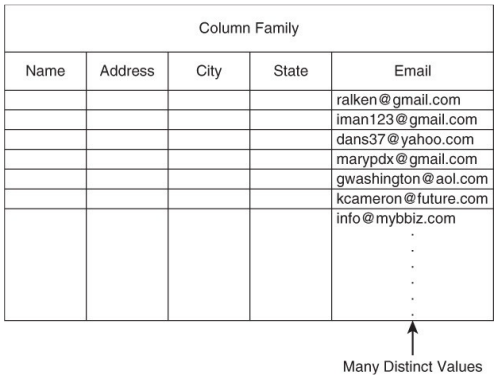
- ***Primary indexes*** – Gli "indici primari" si riferiscono alle *row keys*.
- ***Secondary indexes*** – Gli "indici secondari" si riferiscono ad uno o più *column value*.

Entrambe le tipologie di indici sono create e mantenuti dal DBMS: in generale, è preferibile utilizzare indici creati e mantenuti dal DBMS in modo tale da non doverci preoccupare delle operazioni di creazione e aggiornamento degli indici ogni volta che una nuova *row* viene inserita o una row esistente viene modificata.

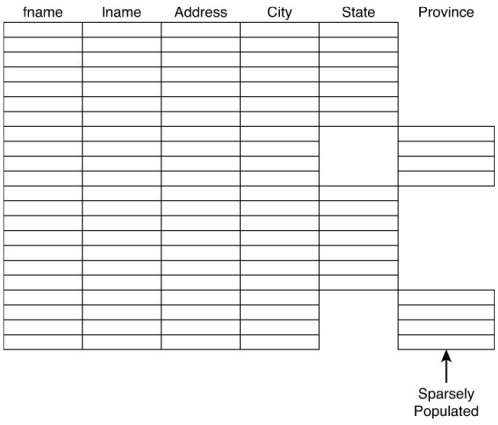
Quando è sconsigliato fare uso degli indici?



Scenario (1): istanziare un' indice associato ad una colonna i cui valori sono di due tipi (esempio: true o false) ed equamente distribuiti in numero.



Scenario (2): istanziare un indice riferito ad una colonna i cui valori sono diversi, in generale, da row a row.



Scenario (3): istanziare un indice su una colonna popolata in maniera sparsa.

Questions

How are associative arrays different from arrays? (*key – value DB*)

Un array è una struttura dati; un array assume la forma di una lista ordinata di coppie <chiave, valore>. Negli array associativi la chiave può essere di qualsiasi tipo mentre in un array la chiave è un intero. Inoltre, i valori memorizzati in un array sono omogenei (ovvero tutti i valori presenti nell'array sono dello stesso tipo) mentre, negli array associativi, i valori possono essere eterogenei tra loro. In particolare, negli array associativi, si parla di valori come *oggetti*: un oggetto può essere un intero, un numero con la virgola, una stringa, un oggetto JSON, eccetera. Da ciò si deduce che sia le chiavi sia i valori memorizzati da un array associativo possono essere eterogenei.

Esempio array:

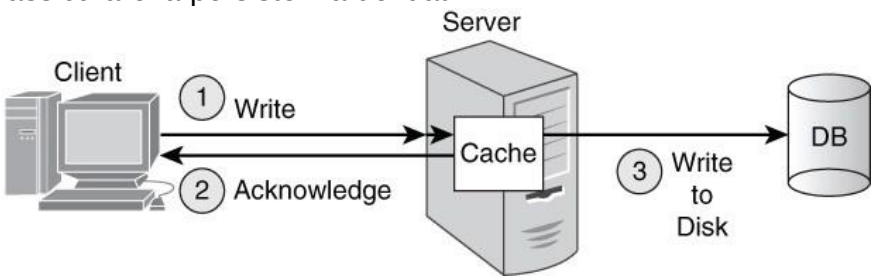
```
array[0] = 10
array[1] = 12
array[2] = 14
array[3] = 16
array[4] = 18
```

Esempio array associativo:

```
array['first element'] = "Hello"
array[1] = { "text" : "World" }
array["third element"] = 3
```

How can you use a cache to improve relational database performance? (*key – value DB*)

Il meccanismo di caching è spesso utilizzato per aumentare le performance in termini di diminuzione dei tempi di risposta. In particolare, tutti i database devono essere mantenuti su un dispositivo di memorizzazione persistente per assicurare la persistenza dei dati. Ciò è svantaggioso in termini di tempo di risposta perché l'accesso (lettura/scrittura) al disco è costoso. Il meccanismo del caching prevede di portare i record del DB dalla dispositivo persistente alla memoria RAM.
Possiamo notare che questo approccio, oltre ad essere vantaggioso durante le operazioni di lettura, permette di velocizzare le operazioni di scrittura. Supponiamo che l'applicazione necessita di eseguire un operazione di scrittura su un dato. Tale operazione può essere eseguita direttamente sulla cache (quindi in memoria RAM con un notevole risparmio di tempo). Una volta conclusa la scrittura in cache il database restituisce il controllo all'applicazione. In seguito, l'operazione di scrittura viene eseguita anche sul dispositivo di memorizzazione persistente per assicurare la persistenza dei dati.



Si noti che, nel caso in cui si verificasse un errore prima che l'operazione di scrittura venga eseguita anche sul dispositivo di memorizzazione persistente, allora l'operazione viene persa per sempre causando *inconsistenza*.

What is a namespace? (*key – value DB*)
Un namespace è un insieme di coppie <chiave, valore> il cui unico requisito necessario è che ogni chiave sia univoca all'interno del namespace. Di fatto, possono coesistere due chiavi uguali a patto che appartengono a namespace diversi.

Describe a way of constructing keys that captures some information about entities and attribute types. (*key – value DB*)
Visto che nei KV DB la chiave è l'unico strumento utile per accedere al dato oggetto di ricerca, e visto che i KV DB non supportano dei linguaggi per eseguire queries, è utile definire delle chiavi significative (noti qualche differenza con le chiavi dei database relazionali?). In particolare, nella fase di design, lo sviluppatore deve prevedere un pattern che ogni chiave deve rispettare. Un esempio utile può essere il seguente:

entityName:entityID:entityAttribute

Se le chiavi sono costruite rispettando il pattern sopra definito allora è possibile definire dei metodi *generali* che ci permettono di recuperare/settare il valore associato ad un chiave. Esempio:

```
define setCustomerAttribute(string customer_id, string attribute, string value):
    string key = "cust:" + customer_id + ":" + attribute
    customer[key] = value

define getCustomerAttribute(string customer_id, string attribute):
    string key = "cust:" + customer_id + ":" + attribute
    return customer[key]
```

Name three common features of key-value databases. (*key – value DB*)

- Semplicità
- Velocità
- Facilità di scaling

What is an anti – entropy process? (*column DB*)
Un anti – entropy è un processo atto a scovare le differenze tra repliche. Perché è necessario verificare che le repliche non siano differenti tra loro? La differenza tra repliche è sinonimo di ***inconsistenza***! Vediamo un esempio di implementazione:

- L'anti – entropy process si basa sugli *hash tree*. Un hash tree è una struttura dati, in particolare un albero, con le seguenti caratteristiche:
    - Le foglie dell'albero hanno come label l' hash di un sottoinsieme dei dati che compongono il database.
    - I nodi hanno come label l' hash di tutti i figli. In particolare, un nodo appartenente al livello *i* ha come label l'hash di tutti i livelli successivi a partire da *i + 1* (compreso).
- Ogni nodo del cluster calcola il proprio hash tree e lo scambia con gli altri nodi appartenente al cluster. Se due hash tree sono diversi questo significa che anche le due repliche sono diverse (ciò è dovuto al fatto che, a meno delle collisioni, le funzioni hash sono iniettive).
- Il confronto tra due hash tree può essere compiuto direttamente sulla radice: se coincidono possiamo affermare che le due repliche sono uguali. In caso di esito negativo si prosegue il confronto con i nodi di secondo livello. Il confronto prosegue seguendo il

percorso dei nodi che hanno dato esito negativo: seguendo questo percorso si arriva alla foglia, dunque al sottoinsieme dei dati, che differisce tra i due nodi.

Questo approccio è vantaggioso in termini di spazio e risorse utilizzate visto che il processo si basa sullo scambio di messaggi di dimensione piccola (un hash tree occupa molto meno spazio rispetto all'intera replica) e le operazioni atte a risolvere l'inconsistenza sono effettuate solo su un sottoinsieme dei dati.

How to represent a one-to-many relationship in document databases. (*document DB*)

Nei database relazionali una relazione uno a molti si traduce attraverso il meccanismo delle chiavi esterne. In particolare, dal corso di basi di dati ci dovremo ricordare che la traduzione di una relazione one-to-many avviene con l'aggiunta della chiave primaria dell'entità lato uno ad ogni row dell'entità lato molti.
Facciamo un esempio: un ordine è associato ad un solo cliente ed un cliente può avere fatto più ordini.

Customer(***customer_id**, first_name, last_name, address, payment_method*)
Order(***order_id**, date, delivered, **customer_id**)*

Se invece abbiamo deciso di utilizzare un document DB abbiamo due possibili soluzioni: *document embedding*, *document linking*.

- ***Document embedding*** – Questa soluzione consiste nell' inserire un array di documenti in un campo del documento che modella l'entità lato uno. Tale array è composto, per l'appunto, da documenti che modellano l'entità molti. Attenzione: se dell'entità lato molti esiste una collection allora stiamo introducendo ridondanza! Dobbiamo essere consapevoli che questo metodo porta a dover gestire l'aggiornamento (se non si vuole incorrere in problemi di inconsistenza) di due collection ogni qual volta che un documento della collection che modella l'entità lato molti viene modificato.
- ***Document linking*** – Questa soluzione consiste nell'inserire un array di id nel documento che modella l'entità lato uno. Gli id presenti nell'array sono gli id associati ai documenti che modellano l'entità lato molti. Questa soluzione ricalca il modello relazionale! Esempio: se vogliamo vedere tutti gli ordini fatti da un certo cliente dobbiamo prima recuperare la lista di id presente nel documento del cliente e poi accedere *n* volte alla collection *Order* per recuperare ogni ordine fatto dall'utente.

Replication (*miscellaneous*)
Uno dei non-functional requirements ricorrenti in questi ultimi tempi è la *high availability*. Un modo per assicurarci che il database sia sempre *raggiungibile* è duplicare l'intero set di dati su più server. Noi abbiamo studiato due paradigmi:

- *Master slave replication* – L'architettura è composta da un nodo, detto *master*, che ricopre un ruolo centrale all'interno del cluster. Quest'ultimo è l'unico a ricevere e gestire le operazioni di scrittura. Tutti gli altri nodi, detti *slave*, accettano operazioni di lettura. Ogni nodo del cluster, ovviamente, possiede la replica del dataset. Il master dataset è mantenuto dal master; si noti che è il master a dover propagare le operazioni di modifica a tutti gli slave altrimenti, questi ultimi, avrebbero una versione non aggiornata del dataset. Uno dei vantaggi di questa architettura è la *semplicità*: le operazioni di scrittura non necessitano di alcuna coordinazione visto che è il master a gestire la propagazione di queste ultime. Purtroppo, il master costituisce un *single point of failures*. Nell'architettura originale sono presenti solo le "connessioni" tra il master e gli slaves. Se aggiungiamo anche le connessioni tra slaves è possibile mitigare il problema del

single point of failures. Esempio: periodicamente il master invia un messaggio a tutti gli slaves; se dopo un certo periodo di tempo gli slaves non ricevono più il messaggio dal master assumono che quest'ultimo sia caduto ed eseguono un protocollo per l'elezione di un nuovo master tra gli slaves.

- *Masterless replication* – L'architettura è strutturata, da un punto di vista logico, secondo una topologia a ring. Tutti i nodi del cluster possono ricevere operazioni di lettura e scrittura. Ovviamente, visto che tutti i nodi possono gestire operazioni di lettura e ci sono più repliche del dataset, si deve gestire la consistenza tra repliche. Esempio: quando un nodo esegue un operazione di scrittura, quest'ultimo propaga l'aggiornamento ai suoi nodi "vicini" (il precedente ed il successivo in senso circolare).

Where can we use hashing in distributed databases? (*document DB*)
Nel contesto dei database distribuiti l'espressione *"hash function"* può essere incontrata quando si parla di ***data partitioning*** *(**sharding**)*.

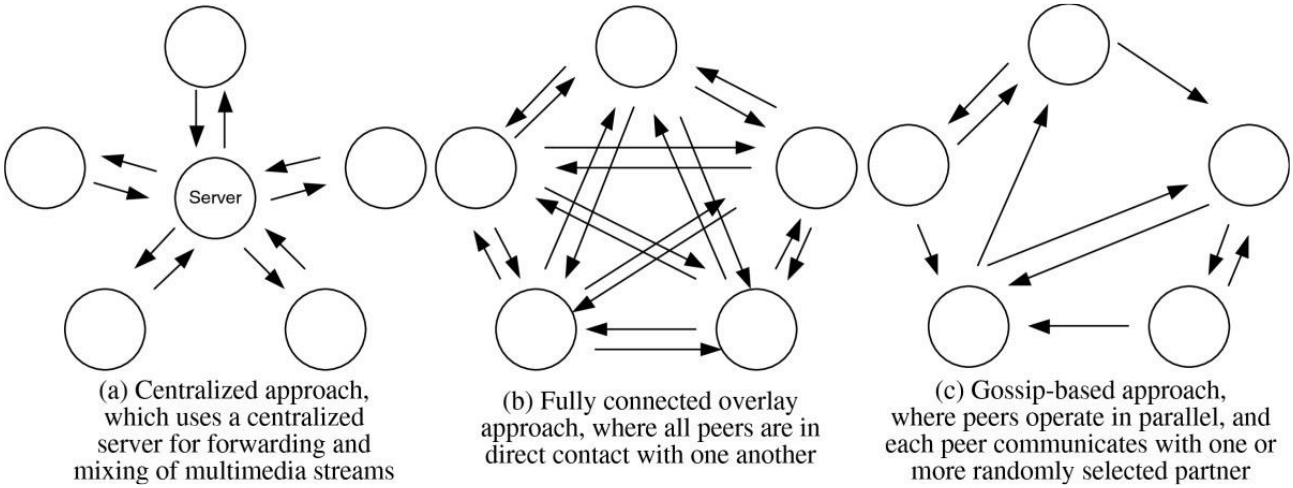
What is hotspotting and how to avoid it? (*column DB*)
Si verifica *hotspotting* quando le operazioni di scrittura sono frequentemente dirette ad un sottoinsieme di nodi del cluster (nel caso peggiore ad un solo nodo). Ciò significa che il workload non è bilanciato equamente tra i nodi del cluster. Se prendiamo in esame l'architettura HBase, i dati sono memorizzati ordinati, in maniera lessicografica, secondo la row key. È probabile che la row key sia generata in maniera incrementale e di conseguenza, durante il popolamento del DB, è possibile che un server alla volta gestisca da solo il carico del popolamento. Per far si che ciò non accada è possibile sfruttare le *hash function*, facendo l'hash della row key, oppure aggiungendo un prefisso casuale alla row key. Entrambe le soluzioni permettono di distribuire equamente il carico su tutti i nodi del cluster.

What is the difference between OLTP and OLAP? (*miscellaneous*)
Un applicazione è detta OLTP, OnLine Transaction Processing, quando compie un numero considerevole di operazioni CRUD. D'altro canto, un applicazione è detta OLAP, OnLine Analityc Processing, quando compie analisi sui dati.

Why do we need a commit log in a columnar database? (*column DB*)
Come ben sappiamo, la particolarità dei columnar DB è data dal fatto che una colonna è memorizzata interamente all'interno di un blocco. Questo aspetto, seppur vantaggioso quando si tratta di eseguire analisi sui valori di un attributo (quindi di una colonna), è causa di pessime perfomance quando si tratta di applicazioni con scritture frequenti. Infatti, quando si deve inserire/aggiornare una row è necessario accedere a svariati blocchi del dispositivo di memoria persistente.
Per mitigare questo problema è possibile fare uso di file, detti *commit log*. Tali file sono *append only* e risiedono in memoria RAM (quindi le operazioni di scrittura avvengono in tempi brevi). Ogni qual volta l'applicazione richiede l'esecuzione di una scrittura, il DBMS appende un record sul commit file con tutte le informazioni necessarie per eseguire l'inserimento/aggiornamento del/dei dato/i. Una volta scritto il commit log, il DBMS restituisce il controllo all'applicazione. La scrittura in memoria persistente avviene solamente dopo che il controllo viene restituito all'applicazione. Questo approccio, oltre ad una diminuzione dei tempi di risposta, è vantaggioso anche in termini di recupero in caso di failures: il commit log di fatto riporta lo

storico delle modifiche apportate ai dati; ripercorrendo in ordine causale le scritture e possibile ripristinare correttamente lo stato del database.

What are the mainly used communication protocol in a cluster? (*miscellaneous*)



(a) Centralized approach, which uses a centralized server for forwarding and mixing of multimedia streams

(b) Fully connected overlay approach, where all peers are in direct contact with one another

(c) Gossip-based approach, where peers operate in parallel, and each peer communicates with one or more randomly selected partner

---

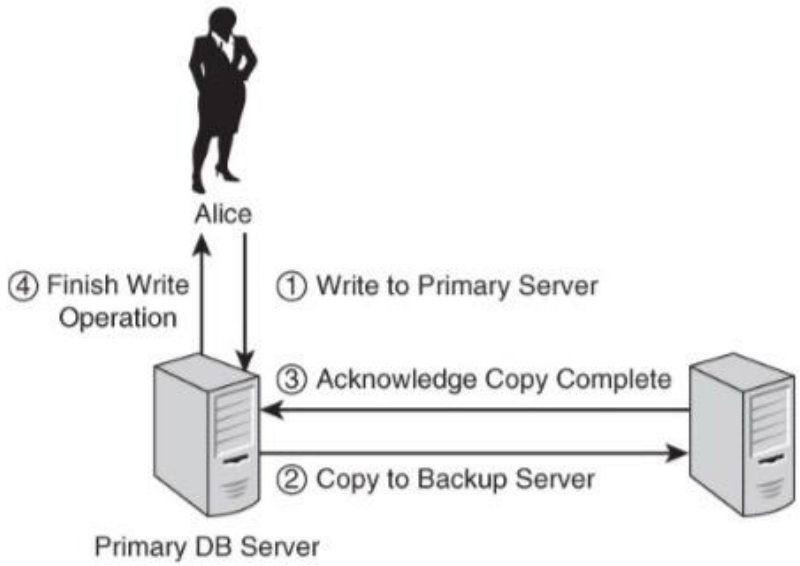
What is the *delta store?* (*column DB*)

Problema: l'architettura base dei column family DB non è grado di gestire efficientemente stream costanti di operazioni di modifica che, nel dominio dei database relazionali, riguardano l'inserimento/modifica di una riga della tabella.

Soluzione: il ***delta store*** è un'area del database mantenuta in memoria avente la caratteristica di non essere compressa e di essere ottimizzata nel gestire operazioni frequenti di modifica dei dati.

Ovviamente, questo tipo di soluzione, porta ad avere dei dati non aggiornati se le operazioni di merge tra delta store e column DB non vengono fatte frequentemente. Per eseguire le analitycs potrebbe essere necessario dover accedere al delta store per avere i dati aggiornati.

What is the TWO-PHASE COMMIT? (*intro*)



Nello scenario dei database distribuiti, visto che i dati sono replicati su più server, è necessario gestire la consistenza della repliche quando si eseguono operazioni di modifica dei dati. Per semplicità, supponiamo di avere un DB su due repliche: un server *master* ed un server di *backup*. Tutte le modifiche apportate al server master devono essere propagate al server di backup e, affinché non insorgano problemi di inconsistenza, tali modifiche devono essere propagate nella stessa transazione. Ecco che entra in gioco la ***two-phase commit***:
Si presti attenzione al fatto che, a fronte di un guadagno di *availability*, dobbiamo mettere in conto una perdita in termini di *latenza*. Inoltre, il sistema non è in grado di rispondere alle queries durante una *two-phase commit*.

What is an ACID transaction? (*intro*)
Una transazione è detta ACID se gode delle seguenti proprietà:

- ***Atomicity*** – Una transazione è atomica quando le sue operazioni sono indivisibili, ovvero tutte le operazioni della transazione sono eseguite sulla base di dati oppure nessuna.
- ***Consistency*** – Una transazione deve lasciare la base di dati in uno stato consistente. La base di dati deve essere in uno stato consistente sia prima sia dopo l'esecuzione della transazione, indipendentemente dall'esito della transazione.
- ***Isolation*** – Una transazione deve essere eseguita in modo isolato, ovvero il risultato della transazione non deve essere influenzato dall'esecuzione di altre transazioni e viceversa.
- ***Durability*** - Le modifiche apportate alla base di dati dalla transazione devono essere persistenti nel tempo, anche in caso di *failures*.

State what BASE stands for? (*intro*)
Se i database relazionali godevano delle proprietà ACID, i database NoSQL godono delle proprietà BASE:

- ***Basically Available*** – *Basically Available* significa che la base di dati, anche nel caso di system failures parziali o network partition, è raggiungibile dall'applicazione. Di conseguenza il servizio offerto dall'applicazione supportata da un NoSQL DB è, in pratica, sempre disponibile.
- ***Soft state*** – I dati memorizzati potrebbero non essere aggiornati all'ultima versione a causa dell'*eventual consistency model*.
- ***Eventual consistency*** – Tale modello ci assicura che, *eventually* e quindi *prima o poi*, tutti le repliche convergeranno alla stessa versione dei dati.

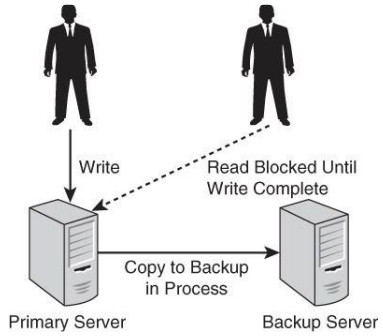
Grazie a questo modello di consistenza è possibile assicurare *high availability* e *low response times*. Poco sopra abbiamo studiato la *two-phase commit* che ci assicurava consistenza anche nello scenario in cui la base di dati era replicata su più server. Ovviamente, la *two-phase commit* è più costosa (in termini sia di latenza sia di dati scambiati in rete) rispetto alla controparte utilizzata in presenza di una sola istanza della base di dati. Grazie al modello introdotto con l'*eventual consistency*, le transazioni ora sono meno "**restrittive**".

What does it mean that *Consistency*, *Availability* and *Partition Tolerance* cannot be ensured all at the same time? (*intro*)
Facciamo alcuni esempi per convincerci che la domanda dice la verità.

Esempio (1): supponiamo di gestire un sito di e-commerce e supponiamo di utilizzare una base di dati NoSQL che supporta il modello *eventual consistency*. Consideriamo il caso d'uso

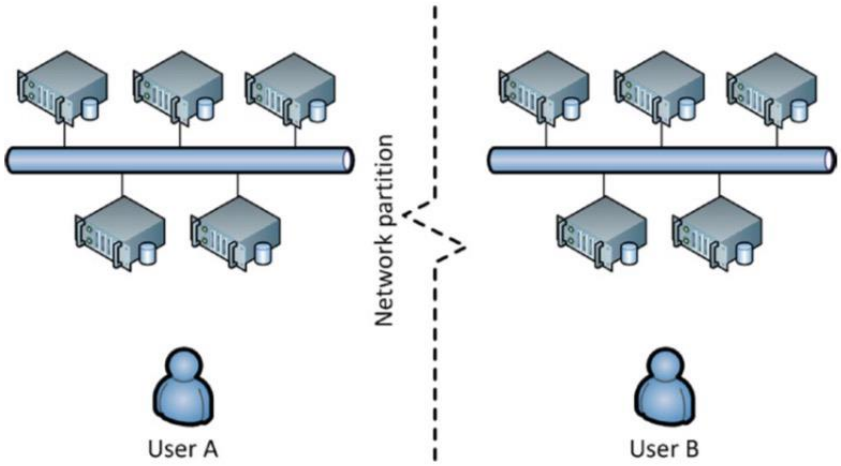
del carello: se le scritture sono gestite secondo un modello di *eventual consistency* allora il DBMS restituisce il controllo all'applicazione una volta che ha eseguito l'operazione di modifica nella sua replica e solo dopo propaga la modifica sulle altre repliche. Dunque, esiste un lasso di tempo in cui il server "principale" e il server di "backup" possiedono due versioni diverse dei dati. In questo caso abbiamo *availability* ma non *consistency*.

Esempio (2): questo scenario si basa sul modello di consistenza offerto dalla *two-phase commit*. Affinché la consistenza sia assicurata, le operazioni di lettura eseguite durante una *two-phase commit* possono essere bloccato con lo scopo di bloccare le letture di dati spuri.



Esempio (3): supponiamo che il nostro cluster sia stato diviso da una network failures e che dunque le nostre repliche siano divise in due partizioni che non si possono raggiungere. In questo scenario possiamo comportarci sostanzialmente in due modi:

1. Permettiamo le letture versioni diverse dello stesso dato → NO *Consistency.* Ciò è causa della partizione della rete: è possibile che il dato che l'user A modifica possa essere replicato in un server dell'altra partizione e di conseguenza, quando l'user B lo legge, è possibile che ottenga una versione non aggiornata visto che non è stato possibile compiere l'aggiornamento.
2. Spegniamo una delle due partizioni fino a quando si ripristina il collegamento tra le due partizioni → NO *Availability*.



CAP Theorem. (*intro*)
Il CAP Theorem afferma che i database distribuiti non possono assicurare *Consistency*, *Availability* e *Partition Tolerance* tutte contemporaneamente. In particolare, al massimo due di queste features possono essere assicurate contemporaneamente.
Vediamo le possibili configurazioni:

- ***CP*** – ***Consistency*** and ***Partition Tolerance***. Questo tipo di configurazione è spesso utilizzata per applicazioni che possono tollerare la latenza di risposta ma non possono tollerare la non consistenza dei dati perché porterebbe perdite, ad esempio, economiche come nel caso di applicazioni finanziarie.
- ***AP*** – ***Availability*** and ***Partition Tolerance***. Questo tipo di configurazione è utile per tutte quelle applicazioni che necessitano di una bassa latenza e di offrire un servizio costantemente raggiungibile. Si pensi ad Amazon: se il servizio non è raggiungibile 24/7 oppure il tempo di risposta è alto, i clienti potrebbero smettere di utilizzare il servizio e diventare clienti di un altro sito di e-commerce.
- ***CA*** – ***Consistency*** and ***Availability***. Questo tipo di configurazione può essere raggiunta con un cluster di server connessi tra loro. In questo modo, a fronte di un aumento della latenza, la consistenza può essere assicurata grazie, ad esempio, a modelli di consistenza come la *two-phase commit*. Inoltre, vista la ridondanza introdotta dal cluster, anche l'availability è assicurata. Ovviamente, se si verifica una network failure, parte dei servizi offerti dall'applicazione possono non essere disponibili visto che non abbiamo la certezza di aggiornare tutte le repliche a causa propria del partizionamento della rete.

Type of Eventual Consistency. (*intro*)

Esistono diverse tipologie di *eventual consistency*:

- ***Read-Your-Writes Consistency*** – Questo modello di consistenza assicura che un utente che esegue delle richieste di lettura, otterrà i dati aggiornati all'ultima versione da lui/lei modificata. Ciò è assicurato in qualunque momento l'utente accede al servizio offerto dall'applicazione.
- ***Session Consistency*** – Questo modello di consistenza assicura che l'utente legga i dati aggiornati all'ultima versione da lui/lei scritta relativamente ad un sessione. Quindi, in sostanza, questo modello di consistenza implementa la *read-your-writes consistency* in una sessione.
- ***Read Monotonic Consistency*** – Questo modello di consistenza assicura che, quando un dato viene letto, le successive letture su quel dato (da parte di qualsiasi entità che interagisce con il DB) non restituiranno una versione precedente.
- ***Write Monotonic Consistency*** – Questo modello di consistenza assicura che, quando un dato viene modificato da un "utente" con una sequenza di scritture allora le operazioni di scrittura saranno eseguite secondo l'ordine con il quale sono stato inviate.
- ***Casual Consistency*** – Questo modello di consistenza assicurare che le "scritture" avvengono seguendo un ordine *causale*. Facciamo un esempio: supponiamo di gestire i post e i commenti di un social network utilizzando la *casual consistency* come modello di consistenza. Supponiamo di avere la seguente sequenza di post e commenti:

  post(1) Federico: Ciao a tutti, come va?
  commento(1.1) Angelo: Ciao Fede, io e Desi tutto bene 😊.
  post(2) Caterina: Ciao, qualcuno sa dirmi dove mangiare un buon Ramen?
  commento(2.1) Federico: Prova da Miu Fish 😊.
  post(3) Luca: Domani mi laureo, SIUUUUM!
  commento(3.1) CR7: Muchas gracias aficion, esto es para vosotros!

  Una possibile sequenza di scritture corretta:

  post(1)
  commento(2.1)
  post(3)
  commento(3.1)
  post(2)
  commento(2.1)

Una sequenza di scrittura scorretta:

post(1) → Non esiste una relazione di causalità tra
commento(2.1) → queste due scritture
…

Labs

MongoDB

MongoDB was launched in 2009 as a completely new class of general-purpose database and quickly established itself as one of the most popular databases among developers. MongoDB retains the best aspects of relational and NoSQL databases while providing a technology foundation that enables organizations to meet the demands of modern applications. It does this by ***replacing the rigid tables of relational databases with flexible documents*** that map to the way developers think and code. Instead of storing data in columns and rows, document databases can store data as JSON (JavaScript Object Notation). A document database can store any type of data, and the structure of documents can be easily modified.

The three primary advantages of the document data model are:

1. ***Intuitive***: faster and easier for developers. Documents in the database directly map to the objects in your code, so they are much more natural to work with. The following example of a JSON document in MongoDB demonstrates how a customer object is modeled in a single document structure with related data embedded as subdocuments and arrays. This approach collapses what would otherwise be seven separate parent-child tables linked by foreign keys in a relational database.

```
{
  "_id":
      ObjectId("5ad88534e3632e1a35a58d00"),
  "name": {
    "first": "John",
    "last": "Doe" },
  "address": [
    { "location": "work",
      "address": {
        "street": "16 Hatfields",
        "city": "London",
        "postal_code": "SE1 8DJ"},
      "geo": { "type": "Point", "coord": [
          51.5065752,-0.109081]}},
  ],

  "phone": [
    { "location": "work",
      "number": "+44-1234567890"},
    ],
  "dob": ISODate("1977-04-01T05:00:00Z"),
  "retirement_fund":
        NumberDecimal("1292815.75")
}
```

2. ***Flexible schema***: dynamically adapt to change. A document's schema is dynamic and self-describing, so you don't need to predefine it in the database. Fields can vary from document to document (*polymorphism*), and you can modify the structure at any time, allowing you to continuously integrate new application functionality without dealing with disruptive schema migrations.
3. ***Universal***: JSON documents are everywhere. MongoDB stores data as JSON documents in a binary representation called ***BSON*** (Binary JSON). Unlike most

databases that store JSON data as primitive strings and numbers, the BSON encoding extends the JSON representation to include additional types such as int, long, date, floating point, and decimal128. This makes it much easier for applications using MongoDB to reliably process, sort, and compare data.

Working with document data – MongoDB Aggregation Pipeline allows you to transform and analyse data. Documents enter a multistage pipeline that transforms them into an aggregated result. The most basic pipeline stages provide filters that operate like queries and document transformations that modify the form of the output document. Other pipeline operations provide tools for grouping and sorting documents by specific fields, as well as tools for aggregating the contents of arrays, including arrays of documents. In addition, pipeline stages can use operators for tasks such as calculating an average or concatenating a string. The pipeline provides efficient data aggregation using native operations within MongoDB and is the preferred method for data aggregation in MongoDB.

Availability with replica sets – MongoDB replica sets enable you to create up to 50 copies of your data, which can be provisioned across separate nodes, data centers, and geographic regions. Replica sets are predominantly designed for resilience. If a primary node suffers an outage or is taken down for maintenance, the MongoDB cluster will automatically elect a replacement in a few seconds, switching over client connections and retrying any failed operations for you. The replica set election process is controlled by sophisticated algorithms based on an extended implementation of the Raft consensus protocol.

Write concern – Through MongoDB's write concern, you can ensure write operations propagate to a majority of replicas in a cluster. With MongoDB 5.0, the default durability guarantee has been elevated to the majority (w:majority) write concern (MongoDB, by default, adopts a *strict consistency model*). Write success will now only be acknowledged in the application once it has been committed and persisted to disk on a majority of replicas.

Native sharding – Through native sharding, MongoDB can scale out your database across multiple nodes to handle write-intensive workloads and growing data sizes. Sharding with MongoDB allows you to seamlessly scale the database as your applications grow beyond the hardware limits of a single server, and it does so without adding complexity to the application. You have the flexibility to refine or change your shard key — which determines how data is distributed across a sharded cluster — on demand without impacting system availability. By simply hashing a primary key value, many distributed databases randomly spray data across a cluster of nodes, imposing performance penalties when data is queried or adding application complexity when you need to locate data in a specific region. By exposing multiple sharding policies to developers, MongoDB offers a better approach:

- ***Ranged sharding*** – Documents are partitioned across shards according to the shard key value. Documents with shard key values close to one another are likely to be co-located on the same shard. This approach is well suited for applications that need to optimize range-based queries, such as co-locating data for customers in a specific region on a specific set of shards.
- ***Hashed sharding*** – Documents are distributed according to an MD5 hash of the shard key value. This approach guarantees a uniform distribution of writes across shards, which is often optimal for ingesting streams of time series and event data.
- ***Zoned sharding*** – This allows developers to define specific rules governing data placement in a sharded cluster.

Documents – MongoDB stores data records as BSON documents. BSON is a binary representation of JSON documents, though it contains more data types than JSON.

```
var mydoc = {
               _id: ObjectId("5099803df3f4948bd2f98391"),
               name: { first: "Alan", last: "Turing" },
               birth: new Date('Jun 23, 1912'),
               death: new Date('Jun 07, 1954'),
               contribs: [ "Turing machine", "Turing test", "Turingery" ],
               views : NumberLong(1250000)
             }
```

- The field name **_id** is reserved for use as a primary key; its value must be unique in the collection, is immutable, and may be of any type other than an array.

The _id Field – In MongoDB, each document stored in a collection requires a unique ***_id*** field that acts as a primary key. If an inserted document omits the ***_id*** field, the MongoDB driver automatically generates an ObjectId for the ***_id*** field.
The _id field has the following behaviour and constraints:

- By default, MongoDB creates a unique index on the ***_id*** field during the creation of a collection.
- The ***_id*** field is always the first field in the documents. If the server receives a document that does not have the ***_id*** field first, then the server will move the field to the beginning.

CRUD operations

Insert a single document – db.collection.insertOne() inserts a *single* document into a collection.

Example:

```
db.inventory.insertOne(
   { item: "canvas", qty: 100, tags: ["cotton"], size: { h: 28, w: 35.5, uom: "cm" } }
)
```

Insert multiple documents – db.collection.insertMany() can insert *multiple* documents into a collection. Pass an array of documents to the method.

Example:

```
db.inventory.insertMany([
   { item: "journal", qty: 25, tags: ["blank", "red"], size: { h: 14, w: 21, uom: "cm" } },
   { item: "mat", qty: 85, tags: ["gray"], size: { h: 27.9, w: 35.5, uom: "cm" } },
   { item: "mousepad", qty: 25, tags: ["gel", "blue"], size: { h: 19, w: 22.85, uom: "cm" } }
])
```

Query documents

To select all documents in the collection, pass an empty document as the query filter parameter to the find method. The query filter parameter determines the select criteria:

```
db.inventory.find( {} )
```

To specify equality conditions, use <field>:<value> expressions in the query filter document:

```
db.inventory.find( { status: "D" } )
```

A compound query can specify conditions for more than one field in the collection's documents. Implicitly, a logical AND conjunction connects the clauses of a compound query so that the query selects the documents in the collection that match all the conditions.

```
db.inventory.find( { status: "A", qty: { $lt: 30 } } )
```

Using the **$or** operator, you can specify a compound query that joins each clause with a logical OR conjunction so that the query selects the documents in the collection that match at least one condition.

```
db.inventory.find( { $or: [ { status: "A" }, { qty: { $lt: 30 } } ] } )
```

In the following example, the compound query document selects all documents in the collection where the status equals "A" **and** *either* qty is less than 30 *or* item starts with the character p:

```
db.inventory.find({
  status: "A",
  $or: [
    { qty: { $lt: 30 } }, { item: { $regex: '^p' } }
  ]
})
```