Q&A Large Scale & Multi-structured Databases

Francesco Taverna

Domande teoriche:

1. Cosa succede in caso di network partition? Che proprietà vengono perse e mantenute?

Risposta:

Contesto:

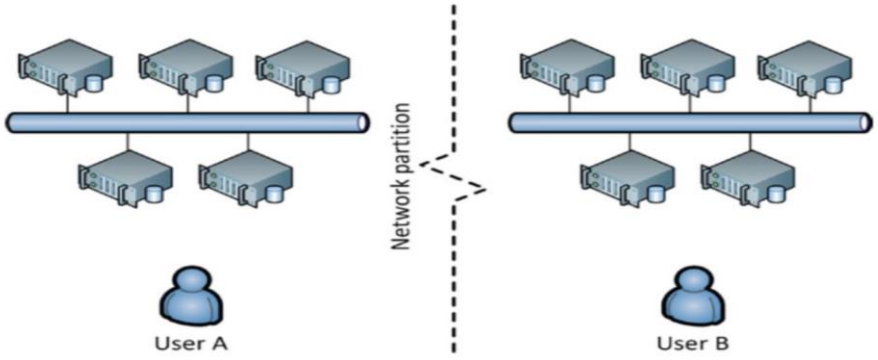
I database in generale permettono di salvare dati e leggerli. I DBMS si occupano di ciò, attraverso 3 principali task:

- Salvare i dati in modo persistente (nessuna perdita quando il database si spegne)
- Mantenere i dati in modo consistente (consistency: tutti i nodi di un sistema vedono gli stessi dati contemporaneamente, se una modifica viene apportata a un dato, tutti i nodi vedono immediatamente il cambiamento)
- Assicurare la disponibilità (availability: i dati devono essere sempre accessibili, bisogna trovare una soluzione ai fallimenti, per esempio avere dei server di backup che intervengono quando uno si rompe)

In generale, queste caratteristiche fanno riferimento ad un contesto distribuito, in molteplici server.

Risposta effettiva:

Il sistema di server distribuiti ha due possibilità: mostrare a ogni utente una diversa visualizzazione dei dati (availability ma non consistency - AP) oppure chiudere una delle partizioni e disconnettere uno degli utenti, isolando alcune porzioni (consistency ma non availability - CP).

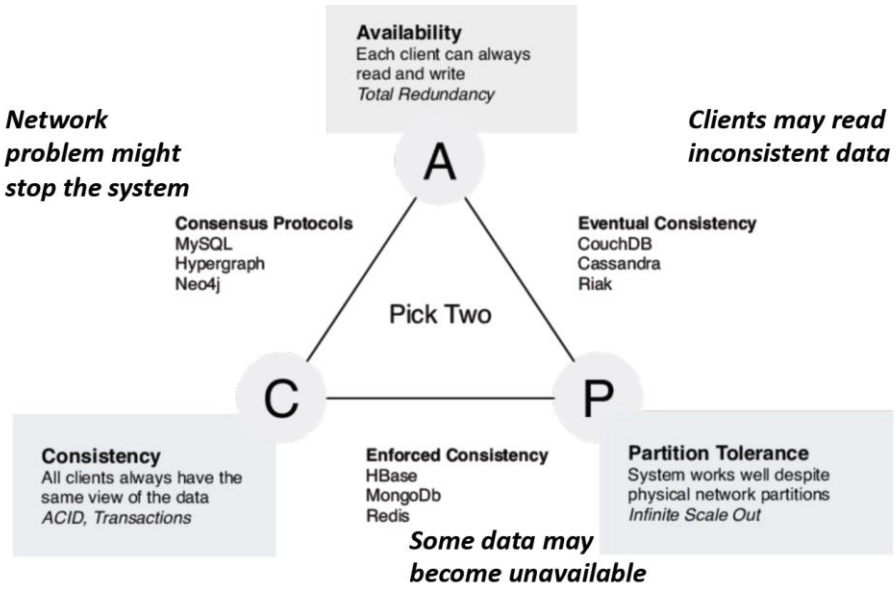


Per capire meglio questo problema introduciamo il teorema CAP.

I database distribuiti non possono garantire allo stesso tempo:

- Consistency (C), la presenza di copie coerenti di dati su server diversi
- Availability (A), ovvero fornire una risposta a qualsiasi query
- Partition protection (P), i guasti dei singoli nodi o le connessioni tra nodi non hanno impatto sul sistema nel suo complesso.

Al massimo **due** delle funzionalità precedenti possono essere trovate in un database distribuito. Lo schema di ciò appena descritto è in figura:



Vediamo le varie soluzioni possibili nel dettaglio:

1. Soluzioni CA:

Nel contesto di un sistema CA, si opta per la **consistency (C)** e la **availability (A)**, ma si è disposti a compromettere la **partition tolerance (P)**. Ciò significa che, in caso di partizione di rete (cioè quando una parte del sistema non può comunicare con un'altra), il sistema **si blocca** per garantire che i dati siano consistenti e disponibili solo quando tutte le parti possono comunicare.

Le soluzioni CA sono più comuni in applicazioni in cui la **consistenza** è fondamentale, come nei sistemi bancari e finanziari, dove è essenziale garantire che le transazioni siano accurate e affidabili. Questi sistemi, ad esempio, sono spesso collegati a database relazionali (RDBMS) che offrono forte consistenza e disponibilità.

Garantire la **consistenza totale** può influenzare negativamente le **prestazioni** del sistema, causando **latenza** e **scalabilità limitata**, perché il sistema deve assicurarsi che tutti i nodi siano aggiornati prima di completare le operazioni, e questo può richiedere tempo.

2. Soluzioni AP

In un sistema distribuito **AP**, è garantita la **availability** e la **partition tolerance**. Tuttavia, durante una partizione di rete, i dati restituiti possono essere **inaccurati**.

Caratteristiche principali:

- **Disponibilità (A)**: Il sistema continua a rispondere alle richieste, anche se non garantisce che i dati siano aggiornati o coerenti.

- **Tolleranza alla partizione (P)**: Il sistema è in grado di funzionare anche se alcune parti non possono comunicare tra loro.
- **Compromesso sulla consistenza (C)**: In questi sistemi, la consistenza dei dati è sacrificata in favore della disponibilità e della tolleranza alla partizione. Ciò significa che il sistema può restituire la versione più recente di un dato che potrebbe essere obsoleta o non sincronizzata con il resto dei nodi.

Note importanti:

- **Dati obsoleti**: Un sistema AP può ritornare la versione più recente dei dati che potrebbe essere "stale" (obsoleta). Può anche accettare scritture che verranno elaborate successivamente, quando la partizione è risolta.
- **Esempi di utilizzo**: Sistemi in cui è essenziale garantire la disponibilità continua, anche in caso di errori esterni, come **carrelli della spesa online** e **sistemi di gestione dei contenuti (CMS) per la pubblicazione di notizie**.
- **Considerazioni:**

Le soluzioni AP sono ideali per applicazioni dove la disponibilità è più importante della consistenza immediata, accettando la possibilità di dati temporaneamente non aggiornati.

3. Soluzioni CP

Le soluzioni **CP** garantiscono la **consistenza** e la **tolleranza alla partizione**. In altre parole, il sistema è in grado di mantenere la coerenza dei dati anche durante la divisione della rete. Tuttavia, durante una partizione, il sistema potrebbe non essere sempre disponibile per rispondere alle richieste.

Caratteristiche principali:

- **Consistenza (C)**: Il sistema assicura che tutte le copie dei dati siano sempre coerenti e aggiornate, quindi, dopo una scrittura, tutte le letture restituiranno la versione più recente dei dati.
- **Tolleranza alla partizione (P)**: Il sistema può continuare a funzionare anche quando ci sono problemi di comunicazione tra nodi.

Applicazioni ideali:

Le soluzioni CP sono adatte a **applicazioni che richiedono coerenza** e in cui un po' di latenza è accettabile. Un esempio comune è il **sistema ATM bancario**, dove è fondamentale che i dati siano aggiornati e coerenti, anche se ciò implica un certo ritardo nella risposta.

Tecnologie tipiche:

Le soluzioni CP si basano su sistemi distribuiti e replicati, sia **relazionali** sia **NoSQL**, che sono progettati per supportare la consistenza e la tolleranza alla partizione.

Parametri di configurazione:

Molti dei framework NoSQL moderni permettono di configurare il livello di **disponibilità** secondo parametri specifici, offrendo la possibilità di bilanciare le esigenze di consistenza e disponibilità in base ai requisiti dell'applicazione.

2. Differenza tra Monotonic, Write e Causal Consistency con contesto.

Risposta:

Contesto: I più recenti NoSQL DBMS sono utilizzati in sistemi distribuiti, ovvero su molteplici server, nei quali si hanno più repliche degli stessi servizi. Tra i pro di questo approccio si ha scalabilità, flessibilità, disponibilità, ma quest'ultima deve essere bilanciata con la consistenza. Per consistency, appunto, si intende che tutti i nodi di un sistema vedano gli stessi dati contemporaneamente, ovvero, se una modifica viene apportata a un dato, tutti i nodi vedono immediatamente il cambiamento attraverso l'invio di aggiornamenti. Mentre con disponibilità (availability) si intende che i dati devono essere sempre accessibili, dunque bisogna trovare una soluzione ai fallimenti, per esempio avere dei server di backup che intervengono quando uno si rompe.

Risposta effettiva: Nei database NoSQL spesso si ricorre all'eventual consistency: ci sarà un periodo di tempo dove le copie dei dati hanno valori diversi, ma prima o poi avranno lo stesso valore. Ci sono vari tipi di questa consistenza, tra le quali abbiamo:

- **monotonic consistency**: si divide in *monotonic write* e *monotonic read consistency*. La write assicura che, se un utente effettua vari aggiornamenti, essi siano eseguiti nell'ordine in cui li ha inviati. La read assicura che, se un utente effettua una query e vede un risultato, nessun'altro utente vedrà mai una versione più vecchia di quel valore.

Esempio read: assumiamo che Alice aggiorni il saldo in sospeso di un customer. Ora il saldo è 1500 euro. Lei lo aggiorna a 2500 euro. Bob legge il saldo e vede 2500. Se Bob legge nuovamente il saldo, vedrà sempre 2500 anche se non tutte le copie hanno questo valore aggiornato.

Esempio write: se un utente ha il saldo in sospeso (da pagare) a 1000 euro e Alice lo riduce del 10%, il saldo è 900 euro. Quell'utente spende 1100 euro, quindi il suo saldo sarà di 2000 euro. Ora però se considero le operazioni in ordine differente ottengo un risultato diverso: se prima l'utente spende, avrà 2100 euro di saldo, poi Alice lo riduce e lo sconto sarà di 210 euro rispetto ai 100 precedenti.

- **read-your-write consistency**: assicura che, se un utente aggiorna un record, tutte le sue letture daranno come risultato il record aggiornato

Esempio: Alice aggiorna il saldo di un cliente a 1500 euro. L'aggiornamento è scritto su un server, il processo di replica inizia ad aggiornare le altre copie. Durante la replicazione Alice legge il saldo, in questo caso si ha garantito che legga 1500 se questo tipo di consistenza è presente.

- **causal consistency**: assicura che gli utenti vedano i risultati in modo consistente con le relazioni causali. Con relazione causale si intende che una operazione dipende causalmente da una operazione precedente.

Esempio: immaginiamo un social network con commenti e risposte

Luca: ho passato l'esame!

Francesco: complimenti!

Marco: da me cè il sole!

Francesco: che fortuna da me piove

Francesco: ho pubblicato un articolo.

Posso vedere i commenti di Luca, Marco e Francesco con ordine differente, ma non posso vedere le risposte in ordine differente, come ad esempio:

Francesco: ho pubblicato un articolo.

Francesco: che fortuna da me piove

Francesco: complimenti!

Marco: ….

Luca: ….

3. Parlami degli indici in MongoDB. A che livello sono definiti? Cosa sono i Compound Indexes?

Risposta:

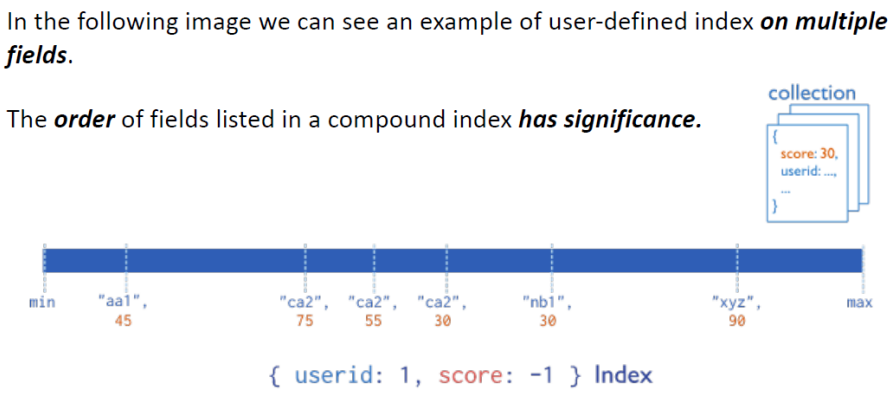
Risposta effettiva:

Gli indici permettono l'esecuzione efficiente delle query in MongoDB. Senza indici, MongoDB deve scansionare ogni documento in una collezione, per selezionare i documenti che corrispondono alla query. Se esiste un indice appropriato per una query, MongoDB può utilizzare l'indice per limitare il numero di documenti che deve ispezionare. Inoltre, MongoDB può restituire risultati ordinati utilizzando l'ordinamento nell'indice. MongoDB definisce gli indici a livello di collezione e supporta gli indici su qualsiasi campo o sottocampo dei documenti in una collezione.

MongoDB crea un indice univoco sul campo *_id* durante la creazione di una collezione. Un indice univoco assicura che i campi indicizzati non memorizzino valori duplicati. L'indice *_id* impedisce ai client di inserire due documenti con lo stesso valore per il campo *_id*.

L'indice sul campo *_id* non può essere eliminato.

Di seguito un esempio di compound index:



Possiamo specificare una query di ordinamento su tutte le chiavi dell'indice o su un sottoinsieme. Tuttavia, le chiavi di ordinamento devono essere elencate nello stesso ordine in cui appaiono nell'indice. Ad esempio, un modello di chiave di indice { a: 1, b: 1 } può supportare un ordinamento su { a:1, b: 1 } ma non su { b: 1, a: 1 }. Affinché una query utilizzi un indice composto per un ordinamento, la direzione di ordinamento specificata per tutte le chiavi deve corrispondere al modello di chiave di indice o corrispondere all'inverso del modello di chiave di indice. Ad esempio, un modello di chiave di indice { a: 1, b: -1 } può supportare un ordinamento su {a: 1, b: -1 } e { a: -1, b: 1 } ma non su { a: -1, b: -1 } o { a: 1, b: 1}.

4. Definizione e differenza tra replicazione e sharding, specificare a cosa servono dal punto di vista delle web scale application.

Risposta:

Contesto: I più recenti NoSQL DBMS sono utilizzati in sistemi distribuiti, ovvero su molteplici server. I pro di questo approccio:

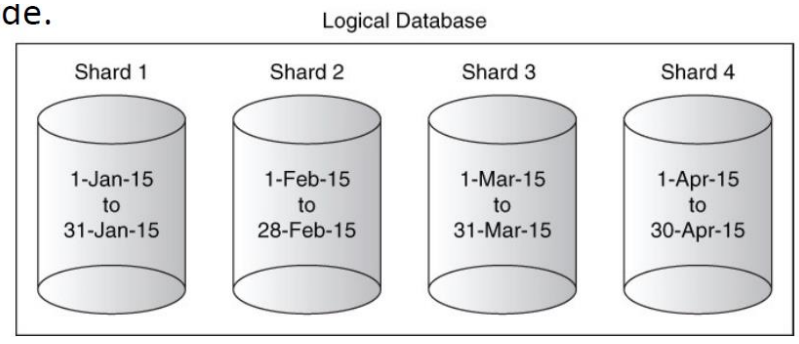
- Assicurano scalabilità, flessibilità, disponibilità
- Facile aggiungere e rimuovere nodi (grazie alla scalabilità orizzontale) invece che quella verticale
- Aggiungono strategie di fault tolerance

Un contro è il bilanciamento tra consistency e availability: per consistency si intende che tutti i nodi di un sistema vedono gli stessi dati contemporaneamente, se una modifica viene apportata a un dato, tutti i nodi vedono immediatamente il cambiamento attraverso invio di aggiornamenti. Mentre con disponibilità si intende che i dati devono essere sempre accessibili, dunque bisogna trovare una soluzione ai fallimenti, per esempio avere dei server di backup che intervengono quando uno si rompe. Ovviamente avere server di backup indica che devono essere aggiornati, quindi qui nasce il conflitto tra le due proprietà.

Risposta effettiva:

Lo sharding o partizionamento orizzontale è il processo di divisione dei dati in blocchi o chunks.

- Ogni blocco, etichettato come shard, viene distribuito su un nodo specifico (server) di un cluster.
- Ogni nodo può contenere solo uno shard.
- In caso di replica dei dati, uno shard può essere ospitato da più di un nodo.



Vantaggi dello sharding:

- Consente di gestire carichi pesanti e l'aumento degli utenti del sistema (load balancing)
- I dati possono essere facilmente distribuiti su un numero variabile di server che possono essere aggiunti o rimossi su richiesta (scalabilità orizzontale)
- Più economico del vertical scaling (aggiunta di RAM e dischi, aggiornamento delle CPU a un singolo server…)
- In combinazione con la replication, garantisce una alta availability del sistema e risposte rapide.

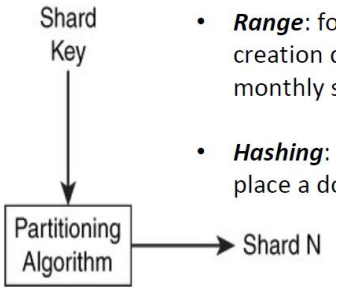
Per implementare lo sharding, i progettisti di document databases devono selezionare una chiave di shard e un metodo di partizionamento. Una chiave di shard è uno o più campi, che esistono in tutti i documenti in una raccolta, che viene utilizzata per separare i documenti.

Esempi di chiavi di shard possono essere: ID documento univoco, Nome, Data, data di creazione, Categoria o tipo, Regione geografica…

In realtà, qualsiasi campo atomico in un documento può essere scelto come chiave di shard.

Esistono tre categorie principali di algoritmi di partizione, in base a:

- Intervallo: ad esempio, se tutti i documenti in una raccolta avessero un campo data di creazione, potrebbe essere utilizzato per suddividere i documenti in frammenti mensili.
- Hashing: una funzione hash può essere utilizzata per determinare dove posizionare un documento. Può essere utilizzato anche un hashing coerente.
- Elenco: ad esempio, immaginiamo un database di prodotti con diversi tipi (elettronica, elettrodomestici, articoli per la casa, libri e vestiti). Questi tipi di prodotto potrebbero essere utilizzati come chiave di frammento per allocare i documenti su cinque server diversi.



La replicazione garantisce una alta availability e consiste nel salvataggio di più copie dei dati nei nodi di un cluster. Il numero di repliche di dati è spesso un parametro da impostare. Maggiore è il numero di repliche, minore è la probabilità di perdere dati, minore sono le performance (in termini di tempo di risposta). Minore è il numero di repliche invece, migliore è la prestazione dei sistemi, ma maggiore è la probabilità di perdere dati.

Un basso numero di repliche può essere utilizzato ogni volta che i dati vengono facilmente rigenerati e ricaricati.

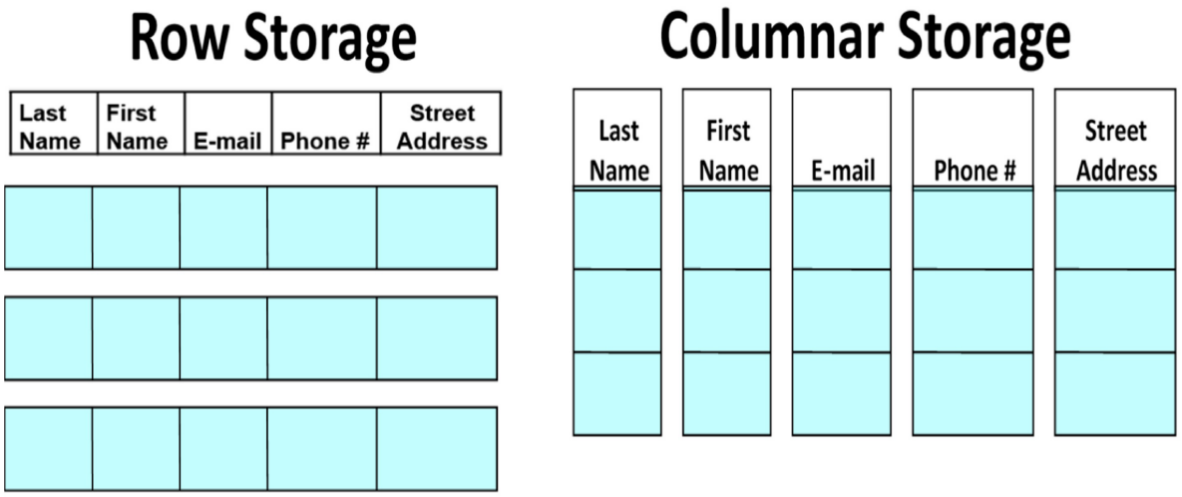
In sintesi le differenze sono:

La replicazione favorisce l'availability, lo sharding il load balancing e la scalabilità orizzontale.

5. Definizione di Keyspace, Row keys, Column family e Column nei Columnar Database

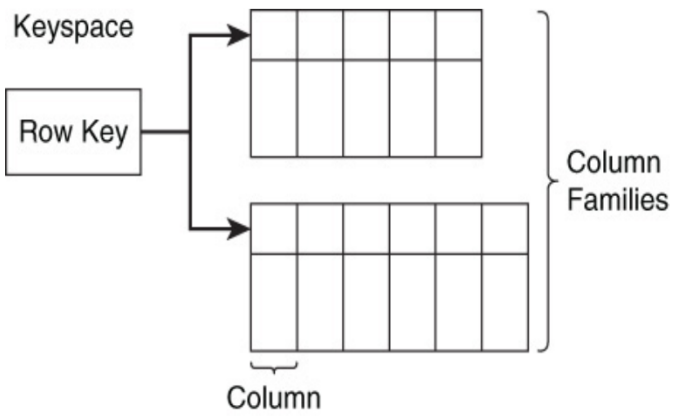
Risposta:

Contesto: Quando elaboriamo dati per fare analisi, nella maggior parte dei casi, non siamo interessati a recuperare tutte le informazioni di ogni singolo record. In effetti, siamo interessati spesso, ad esempio, a recuperare tutti i valori di un attributo di un set di record (per fare grafici di tendenza o calcolare statistiche, ad esempio la media di un attributo). Quando si ha a che fare con l'archiviazione di record basata su righe, dobbiamo accedere a tutti i record del set considerato per recuperare solo i valori di un attributo. Se tutti i valori di un attributo sono raggruppati insieme sul disco (o nel blocco di un disco), l'attività di recupero dei valori di un attributo sarà più veloce di un'archiviazione di record basata su righe.

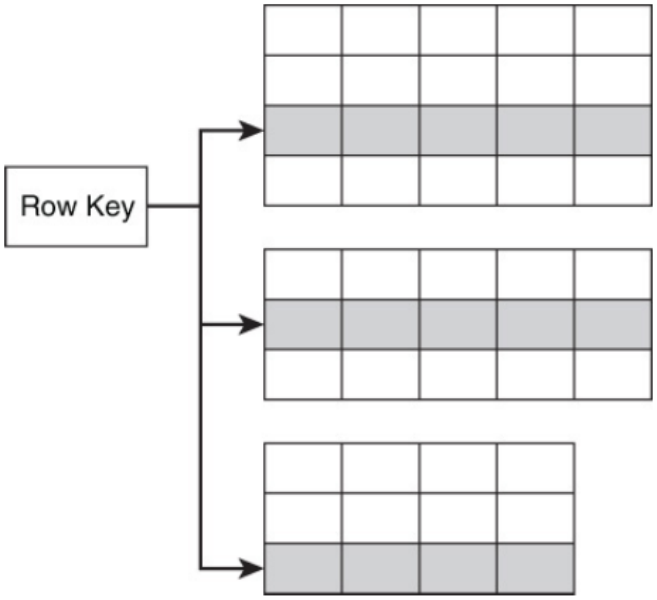


Risposta effettiva:

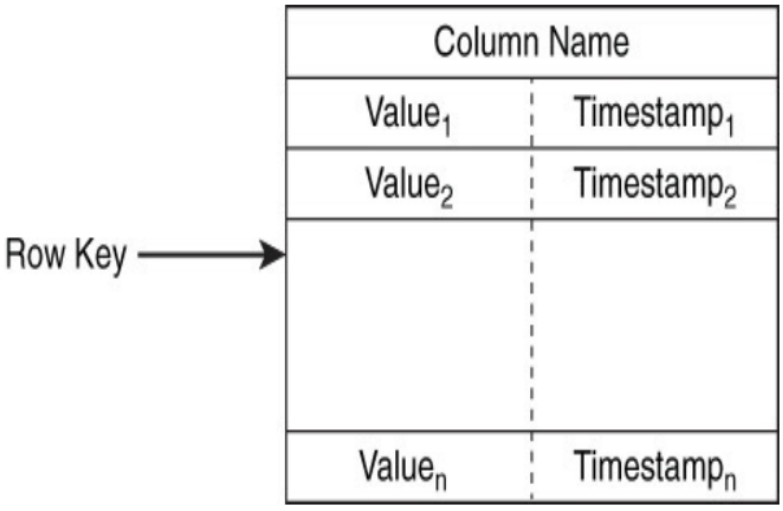
- Il Keyspace è il top-level logical container. Esso salva le Column families, le Row keys e le relative strutture dati. Tipicamente c'è un solo keyspace per applicazione.



- Le Row key sono una delle componenti usate per identificare unicamente i valori salvati in un DB ma sono anche usate per partizionare e per ordinare i dati. Gli algoritmi per il partizionamento e l'ordinamento dipendono dagli specifici database. Ad esempio, Cassandra adotta il *partitioner*, ovvero un oggetto specifico che viene utilizzato per ordinare i dati e per distribuire randomicamente le righe nei nodi.



- Le Column, insieme alle Row key e ad un Version stamp, identificano unicamente i valori. I valori possono essere tipizzati o meno. I timestamp o i Version stamp sono un modo per ordinare i valori di una Column. Le Column in una Column family possono essere modificate dinamicamente.



- Le Column che sono usate frequentemente insieme sono raggruppate nella stessa Column family. Le column family sono analoghe alle tabelle nei database relazionali e devono essere definite a priori. Le righe nelle Column family sono ordinate e "versionate".

In generale, nei Columnar database, le modifiche ai dati, come aggiornamenti o inserimenti, non vengono necessariamente applicate direttamente al dato esistente nel database in

tempo reale. Piuttosto, si utilizza un sistema di *references* o puntatori alle modifiche. Questo significa che ci sono modifiche basate su timestamp: ogni modifica è registrata come un nuovo record con un timestamp, e i dati più recenti vengono selezionati quando si interroga il database.

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

6. Anti-entropy process

Risposta:

Contesto: Il livello di consistency si riferisce alla consistency tra copie di dati su diverse repliche, ciò dipende dai requisiti specifici dell'applicazione.

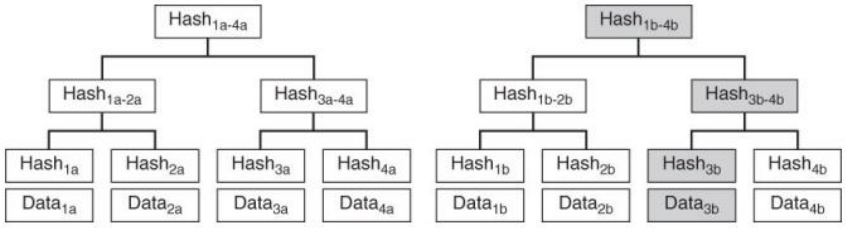
Possiamo considerare diversi livelli di Consistency:

- Strict Consistency: tutte le repliche devono avere gli stessi dati
- Low Consistency: i dati devono essere scritti in almeno una replica
- Moderate Consistency: tutte le soluzioni intermedie

Il livello di consistency è strettamente correlato al processo di replicazione. Il processo di replicazione riguarda dove posizionare le repliche e come mantenerle aggiornate.

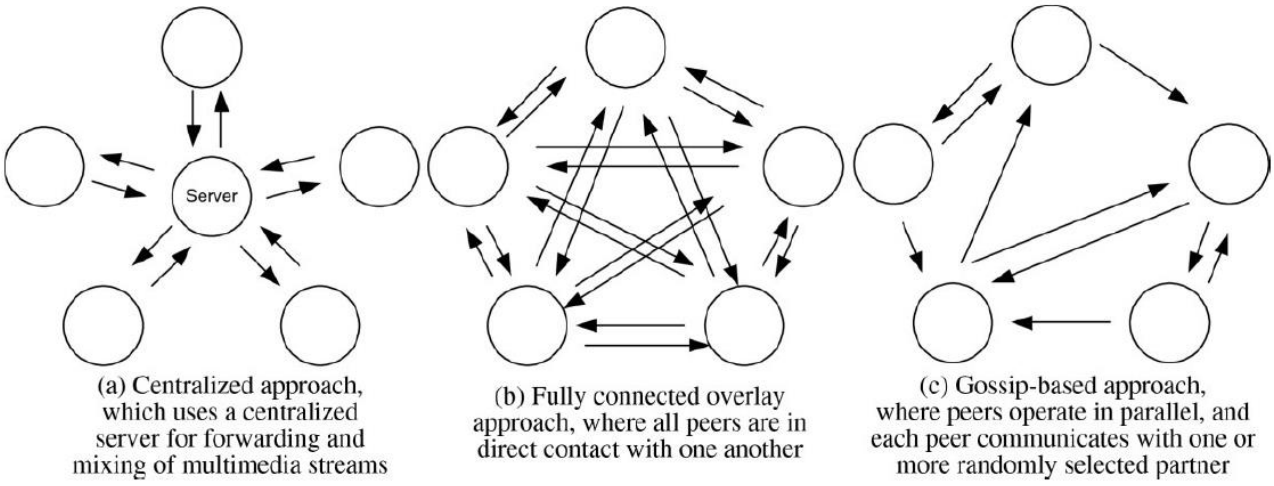
- Di solito, c'è un server predefinito per essere la prima replica che è responsabile di identificare la posizione delle altre repliche tra i server disponibili.
- Il server primario è spesso determinato da una funzione hash.
- Ogni database ha le sue strategie di replicazione.

Risposta effettiva: L'anti-entropia è il processo di rilevamento delle differenze nelle repliche. È importante rilevare e risolvere le incongruenze tra le repliche scambiando una piccola quantità di dati. Gli alberi hash possono essere scambiati tra server per controllare facilmente le differenze perché confrontare un valore hash è molto più veloce che scansionare l'intera partizione. Ogni foglia dell'albero contiene il valore hash di un set di dati nella partizione. Ogni nodo padre contiene il valore hash degli hash dei suoi nodi figli.



1. Un server riceve periodicamente l'albero degli altri nodi.
2. Il server lo confronta con il suo albero.
3. Se le due radici sono uguali: le copie sono identiche e l'algoritmo si ferma.
4. Altrimenti, navigherà nell'albero e confronterà il figlio della radice.
5. Seguendo il nodo diverso, verrà individuata la partizione diversa.
6. Il server con la partizione più vecchia viene aggiornato dall'altro.

Nei sistemi distribuiti, ogni nodo deve essere a conoscenza dello stato degli altri nodi. Nei piccoli cluster, ogni nodo può scambiare messaggi con ogni altro nodo. Il numero totale di messaggi da scambiare per una comunicazione completa è uguale a $n \times (n-1)/2$, dove n è il numero di nodi di un cluster. Di seguito vari protocolli di comunicazione:



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

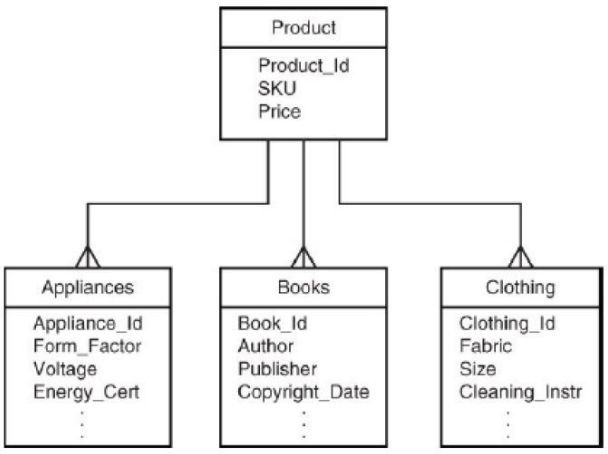
(b) Fully connected overlay approach, where all peers are in direct contact with one another

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

Esercizi pratici

1. Disegnare la tabella con product, book, clothing, appliance e definire: row key, column family e column table. Inserire due record con clothing id e uno con solo appliance ID

Tabella:



Una singola entità, come un particolare cliente o un prodotto specifico, dovrebbe avere tutti i suoi attributi in una singola riga. Ciò può portare a casi in cui alcune righe memorizzano più valori di colonna rispetto ad altre, ma ciò non è raro nei database di famiglie di colonne.

Consideriamo la tabella dei prodotti più in dettaglio. Il rivenditore che progetta l'applicazione vende diversi tipi di prodotti, tra cui elettrodomestici, libri e abbigliamento. Tutti condividono alcuni attributi comuni, come prezzo, unità di stockkeeping (SKU) e livello di inventario. Ognuno di loro ha anche caratteristiche uniche. Gli elettrodomestici hanno fattori di forma, voltaggio e certificazioni energetiche. I libri hanno autori, editori e date di copyright. Gli articoli di abbigliamento hanno tessuti, taglia e istruzioni per la pulizia. Un modo per modellare questo è con diverse tabelle.

In questo caso se utilizziamo un columnar DB si possono memorizzare tutti gli attributi presenti in figura in una sola riga, quindi ci saranno alcune righe che memorizzano più valori colonna di altre.

Per garantire l'atomicità (assicurata a livello di riga) dell'operazione nel columnar DB, la stessa entità nella figura deve essere organizzata in una sola tabella con 4 column families.

Le column families sono: *Product_columns*, *Appliance_columns*, *Clothing_columns* e *Book_columns*. Ogni column family ha gli attributi presenti sopra nell'esempio relazionale.

Il row ID identifica una singola riga di questa tabella. Ogni riga memorizza solo le colonne rilevanti, ad esempio un prodotto di abbigliamento non avrà colonne relative agli elettrodomestici o ai libri. Solo le colonne rilevanti per una certa categoria di prodotto saranno valorizzate, mentre le altre resteranno vuote.

Row ID	Product_columns	Clothing_columns	Appliance_columns
Clothing-1	Product_ID: 101, SKU: C-001, Price: 50.00	Fabric: Cotton, Size: M, Cleaning_Instr: Hand Wash	-
Clothing-2	Product_ID: 102, SKU: C-002, Price: 70.00	Fabric: Wool, Size: L, Cleaning_Instr: Dry Clean	-
Appliance-1	Product_ID: 201, SKU: A-001, Price: 150.00	-	Voltage: 220V, Form_Factor: Compact, Energy_Cert: A+

- Ogni Row ID rappresenta un prodotto specifico.
- Ogni riga utilizza solo le colonne pertinenti per quel tipo di prodotto:

  Clothing riempie solo le colonne di Clothing_columns.

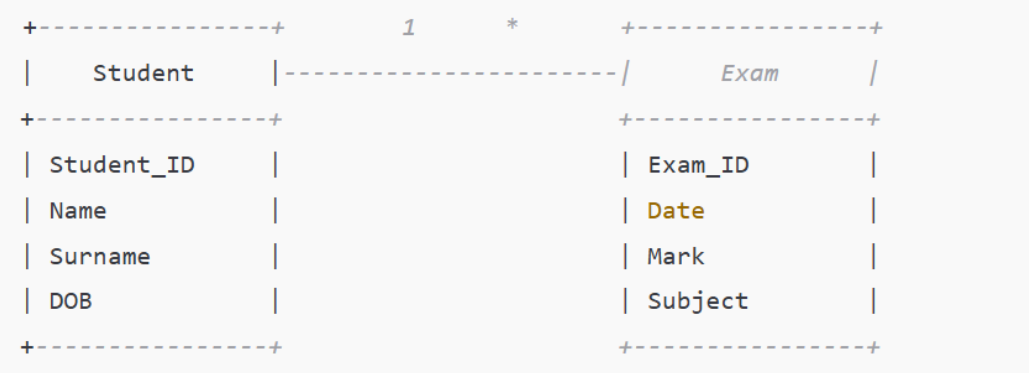
Appliance riempie solo le colonne di Appliance_columns.

- Le famiglie di colonne vuote (es. Appliance_columns per Clothing) rimangono **non memorizzate**, grazie alla natura "sparse" del database.

2. Relazione one-to-many tra studenti ed esami, disegnare l'UML, definire uno o più requisiti funzionali per una soluzione document embedding e stesso per il document linking, scrivere in JSON i documenti per entrambi i casi

1. Let's suppose to consider a one-to-many relationship between the ***student entity*** and the ***exam entity,*** namely one student can be associated to several exams and one specific exam instance is associated to a specific student. The most important attributes for the student entity are: 1) Name, 2) Surname, 3) DOB. As regards the exam entity, the most important attributes are: 1) Date, 2) Mark, 3) Subject . We want to design an application for allowing the student to manage his/her exams.
   We request to:
   a) Depict the UML analysis class diagram
   b) Define some functional requirements supporting the ***document embedding*** strategy for implementing a document DB for the application
   c) Define some functional requirements supporting the ***document linking*** strategy for implementing a document DB for the application
   d) Provide the structure of the collections considering the two alternative document DB implementations, including an example (in correct JSON format) with the documents storing the following information:
      Student1:
      Name: Pippo
      Surname: Pluto
      DOB: 23, Jan 2023
      Exam1:
      Date: 23, Feb 2022
      Mark: 23/30
      Subject: Data Mining
      Exam2:
      Date: 25, Dec 2022
      Mark: 30/30
      Subject: Large Scale

UML:



A. Document Embedding

Tutti i dati di uno studente e dei suoi esami sono memorizzati in un unico documento.

- **Requisiti Funzionali**:

1. Dato uno studente, l'applicazione deve mostrare tutti gli esami che lui/lei ha passato (insieme al voto associato)
2. Dato uno studente, l'applicazione deve consentire di mostrare la media voti dei suoi esami

B. Document Linking

Ogni studente ha un documento separato e ogni esame è memorizzato come un documento separato, collegato tramite una chiave.

- **Requisiti Funzionali**:

1. Dato un esame, l'applicazione deve mostrare la media voti di esso (calcolata tra tutti gli studenti che hanno dato quell'esame)
2. Dato un esame, un mese e una range di voti, l'applicazione deve permettere di calcolare quanti voti per quell'esame sono presenti nel range indicato in quel mese.

Document embedding:


```
{
  "Student_ID": 1,
  "Name": "Pippo",
  "Surname": "Pluto",
  "DOB": "2000-01-15",
  "Exams": [
    {
      "Exam_ID": 101,
      "Date": "2024-11-01",
      "Mark": 28,
      "Subject": "Mathematics"
    },
    {
      "Exam_ID": 102,
      "Date": "2024-11-15",
      "Mark": 30,
      "Subject": "Physics"
    }
  ]
}
```

Document linking:

Studente:

{

"Student_ID": 1,

"Name": "Pippo",

"Surname": "Pluto",

"DOB": "2000-01-15"

}

Esame:

{

"Exam_ID": 101,

"Student_ID": 1,

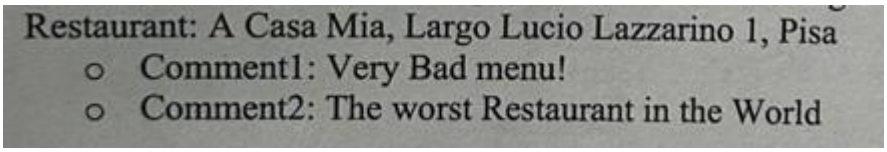
"Date": "2024-11-01",

"Mark": 28,

"Subject": "Mathematics"

}

3. Supponiamo di considerare una relazione one-to-many tra l'entità ristorante e la entità recensioni tale che un ristorante abbia molte recensioni e una recensione sia associata ad un solo ristorante. Discuti in dettaglio come questa relazione può essere implementata in un document DB, fornendo un esempio in formato JSON del DB per ognuna delle possibili implementazioni considerando i seguenti dati:



- **Document embedding**

In questo approccio, tutte le recensioni relative a un ristorante sono incorporate direttamente all'interno del documento che rappresenta il ristorante. È utile quando le recensioni sono strettamente legate al ristorante e devono essere recuperate insieme frequentemente.

{

```
  "restaurant_id": "1",
  "name": "A Casa Mia",
  "address": "Largo Lucio Lazzarino 1, Pisa",
  "reviews": [
   {
    "comment_id": "1",
    "comment": "Very Bad menu!"
   },
   {
    "comment_id": "2",
    "comment": "The worst Restaurant in the World"
   }
  ]
}
```


Vantaggi:

- Migliora le prestazioni quando si recuperano i dati del ristorante insieme alle recensioni.
- Meno operazioni di join, poiché tutti i dati sono nello stesso documento.
- Adatto per un numero limitato di recensioni per ristorante.

Svantaggi:

- Se il numero di recensioni crescesse troppo, il documento potrebbe diventare grande e difficile da gestire.
- Non ideale per query molto specifiche sulle recensioni (ad esempio, cercare quante volte una parola appare nelle recensioni).

- **Document linking**

In questo approccio, i ristoranti e le recensioni sono memorizzati in collezioni separate. Le recensioni contengono un riferimento al ristorante a cui appartengono (ad esempio un *restaurant_id*).

Documento ristorante:


```
{
  "restaurant_id": "1",
  "name": "A Casa Mia",
  "address": "Largo Lucio Lazzarino 1, Pisa"
```

}

Documento recensioni:

[

{

"comment_id": "1",

"restaurant_id": "1",

"comment": "Very Bad menu!"

},

{

"comment_id": "2",

"restaurant_id": "1",

"comment": "The worst Restaurant in the World"

}

]

Vantaggi:

- Più flessibile, poiché le recensioni sono gestite separatamente e possono crescere senza impattare direttamente sul documento del ristorante.
- Utile per query specifiche sulle recensioni o per elaborazioni su recensioni senza caricare l'intero ristorante.

Svantaggi:

- Richiede una sorta di join (applicato tramite codice) per combinare i dati del ristorante con le recensioni.
- Le query che coinvolgono ristorante e recensioni insieme possono essere più lente rispetto all'incorporazione.

4. Considera un namespace, accessibile attraverso la variabile "AppNameSpace" che include le seguenti entrate:

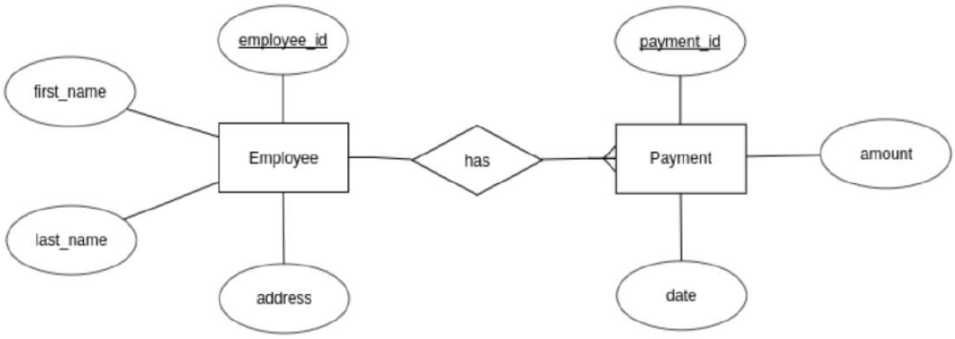
```
cust:1234123:firstName: "Pietro"
cust:1234123:lastName: "Ducange"
```

Definisci i seguenti metodi di set e get:

```
getCustAttr(userId, attributeName)
setCustAttr(userId, attributeName, value)
```

```
public static String getCustAttr(String userId, String attributeName) {
    v_key = "cust:" + userId + ":" + attributeName; // Costruisce la chiave
    return (AppNameSpace[v_key]);
}
public static void setCustAttr(String userId, String attributeName, String value) {
    v_key = "cust:" + userId + ":" + attributeName; // Costruisce la chiave
    AppNameSpace[v_key] = value; // Imposta il valore nel namespace
}
```

5. Traduci relazione 1 a m tra Impiegati e Pagamenti in figura in un document DB. Di seguito sono presenti le due tabelle nel modello relazionale, definisci le query e produci il design del document DB in base alle query che hai definito.



In un modello relazionale possiamo definire le seguenti tabelle. La relazione 1 a molti è gestita usando una chiave esterna:

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

Definizione delle query: Identifichiamo le query principali per capire come strutturare il design del **Document DB**:

- **Query 1**: Recuperare tutti i dettagli di un dipendente dato il suo employee_id.
- **Query 2**: Recuperare tutti i pagamenti effettuati da un dipendente dato il suo employee_id.
- **Query 3**: Trovare il totale dei pagamenti effettuati da un dipendente.

Document embedding: I dati sui pagamenti sono inclusi nel documento del dipendente, perché le query sono orientate a trovare pagamenti a partire da un dipendente.

Collezione dipendenti:

```
{
  "_id": "E001",
  "first_name": "John",
  "last_name": "Doe",
  "address": "123 Main St",
  "payments": [
    {
      "payment_id": "P001",
      "amount": 1500,
      "date": "2024-11-01"
    },
    {
      "payment_id": "P002",
      "amount": 2000,
      "date": "2024-11-15"
    }
  ]
}
```

Le query sono molto efficienti perché tutti i dettagli sono in un unico documento (Nessuna join necessaria):

Se però i pagamenti devono essere frequentemente aggiornati, interrogati separatamente o diventano molto numerosi, l'approccio **con il document linking** è preferibile.

6. Esercizio di seguito

Un'università offre una raccolta di Master Program (MP). Uno studente può iscriversi a un MP. Un MP è composto da un insieme di corsi. Per ottenere il diploma, lo studente deve superare tutti gli esami dei corsi del suo specifico MP. Fornisci il design di un document db (collezioni e struttura) per memorizzare le informazioni relative alla carriera dello studente (inclusi i dati personali) presso l'università e che supporti i seguenti requisiti funzionali:

a. Dato uno studente, l'applicazione deve mostrare tutti i corsi del suo MP.

b. Dato uno studente, l'applicazione deve mostrare tutti gli esami che ha superato (insieme ai voti associati).

c. Dato un MP specifico, l'applicazione deve mostrare la media dei voti degli esami di ciascuno dei suoi corsi.

Si prega di fornire una giustificazione a supporto del design del database e di mostrare un esempio del contenuto del database per il seguente studente:

Nome: Pippo
Cognome: Pluto
Data di nascita: 13 luglio 2001

Master Program: AIDE
Corsi da frequentare: Large Scale, IoT, Cloud Computing, Data Mining
Esami superati: Large Scale (25/30), Data Mining (22/30)

Collezione studente:


```
Fold line
    "name": "Pippo",
    "surname": "Pluto",
    "BOD":"July, 13, 2001",
    "Master Program": "AIDE",
    "Courses": [{"CourseName":"Large Scale", "mark":27},
                {"CourseName":"IoT", "mark":-1},
                {"CourseName":"Data Mining", "mark":22},
                {"CourseName":"Cloud Computing", "mark":-1}
    ]
```


Note:

- Il campo "mark": -1 indica un esame non ancora sostenuto.
- Tutte le informazioni necessarie per le query (a) e (b) sono incluse direttamente in questo documento.

Giustificazione del design del database: La collezione proposta segue un approccio basato sul *document embedding*. Questo approccio è appropriato in questo contesto perché consente di ottimizzare le query (a) e (b) richieste nel problema, che richiedono di accedere rapidamente ai corsi e agli esami associati a uno studente. Includendo tutte le informazioni rilevanti in un unico documento, riduciamo la necessità di operazioni di *join*, migliorando le prestazioni di lettura per query comuni.

Inoltre, si utilizza una struttura separata per i Master Programs (*MP*) con i dettagli dei corsi e la media dei voti (per supportare la query (c)), il che garantisce flessibilità e una chiara separazione delle responsabilità tra collezioni.

Struttura della collezione per i Master Programs (MP):


```
{
  "Name": "AIDE",
```

```
  "Courses": [
    { "CourseName": "Large Scale", “Exams_passed”:[20, 13, 30] },
    …
  ]
}
```

Note:

- Questa collezione contiene i dati globali del master, inclusa la lista dei corsi e la media dei voti ottenuti dagli studenti per ciascun corso.

Vantaggi:

1. **Velocità**: Il design proposto consente di soddisfare le query (a) e (b) direttamente dalla collezione degli studenti senza ulteriori elaborazioni.
2. **Scalabilità**: La collezione dei Master Programs è separata, consentendo aggiornamenti indipendenti ai corsi o ai dati aggregati come la media dei voti.
3. **Flessibilità**: La separazione tra studenti e Master Programs garantisce che le modifiche strutturali in uno non influenzino l'altro.