Sistemi NoSql
DB a Documenti

P. Rullo

Dai relazionali ai NoSql

- Negli ultimi anni si è verificata una esplosione di nuove tipologie di applicazioni legate al web: reti sociali, dispositivi mobili, Internet of Things, ecc.
- Aziende come Twitter, Facebook e Google raccolgono terabyte di dati utente ogni giorno, di tipologia variegata (testi, immagini, ecc.), di struttura variabile e generati ad alta velocità.
- In sintesi, queste le principali caratteristiche di tali dati:
  - Elevato volume (Big Data)
  - Varietà di tipi e formati (testi, immagini, suoni, ecc.)
  - Rapidità dei cambiamenti, anche strutturali.

Dai relazionali ai NoSql

- Si tratta quindi di dati e applicazioni rispetto alle quali la tecnologia “tradizionale” delle basi di dati relazionali risulta inadeguata. E ciò per almeno due classi di motivi:
  - limiti del potere di rappresentazione del modello dei dati
  - limiti architetturali legati alla natura centralizzata dei sistemi relazionali

Dai relazionali ai NoSql

Proprietà tipiche del modello dei dati relazionale sono:

- *Semplicità*. Le relazioni sono insiemi di tuple di dati atomici. Non esistono costrutti idonei a rappresentare e manipolare testi, immagini, ecc.
- *Rigidità della struttura dei dati*. Ogni relazione ha uno schema predefinito, al quale si devono attenere tutte le tuple. Questa rigidità rende difficile uno sviluppo evolutivo della BD, cioè, l'adattamento della BD ai cambiamenti e alle dinamiche che sono tipici di ambienti in evoluzione
- *Normalizzazione*. Ogni relazione memorizza un singolo concetto. Ciò implica una parcellizzazione dell'informazione che limita il livello di ridondanza dei dati e, quindi, l'occupazione di memoria. Prezzo da pagare: inefficienza delle interrogazioni a causa delle numerose operazioni di join che sono necessarie per ricostruire informazioni complessive

Dai relazionali ai NoSql

SELECT *

FROM Cars, Wheels, Seats, Brakes…

WHERE Cars.owner = "Paolo” and Cars.id = Wheels.car_id and Cars.id = Seats.car_id and Cars.id = Brakes.car_id

…

Dai relazionali ai NoSql

- I sistemi NoSql non hanno un unico modello dei dati
- Quattro tipologie di sistemi, classificati in base al modello:
  - sistemi a documenti
  - a grafi
  - basati su valori chiave
  - sistemi a colonne.

Dai relazionali ai NoSql

- A prescindere dallo specifico modello adottato, essi consentono di rappresentare oggetti complessi e, in genere, sono privi di schemi fissi (*schemaless*)
- La possibilità di rappresentare oggetti complessi, permettendo una organizzazione non parcellizzata dell'informazione, favorisce una esecuzione efficiente delle interrogazioni
- L'assenza di schemi rigidi consente quella *flessibilità* necessaria per uno sviluppo veloce e iterativo

Sistemi basati su documenti

NoSql	Relazionale
documento	tupla
Collezione: insieme di documenti	Relazione: insieme di tuple
BD: insieme di collezioni	BD: insieme di relazioni

- Principio: ’pezzi’ di informazione che concettualmente stanno assieme e vengono usati assieme, vanno memorizzati assieme.
- Mentre un sistema relazionale ottimizza i dati dal punto di vista dell’occupazione di memoria, un DMD ottimizza i dati rispetto al modo in cui vengono usati

Rappresentazione di associazioni tramite embedding

- Person(pers_id, first_name, surname, city)
- Car(car_id, model, year, value, owner*)

```
Document persona {
    PersId: 1;
    first_name: “Alvaro”,
    surname: “Ortega”,
    city: “Valencia”,
    cars: [
    { model: “Bentley”,
      year: 1973,
      value: 100000},
    { model: “Rolls Royce”,
     year: 1965,
     value: 330000} ]
    }
```

- In tale documento, l’associazione 1-n è rappresentata incorporando (*embedding*) nell’oggetto che sta dalla parte 1 (persona) tutti gli oggetti dalla parte n (auto).
- Infatti, il documento che rappresenta Alvaro incorpora come sotto-documenti tutte le auto di cui è proprietario.
- L’attributo *cars* è di tipo array di oggetti [c_1, …, c_n], dove ogni oggetto c_i rappresenta una particolare auto.

Rappresentazione di associazioni tramite embedding

- Person(pers_id, first_name, surname, city)
- Car(car_id, model, year, value, owner*)

Document automobile {

car_id: 101;

model: “Bentley”

year: 1973,

value: 100000,

owner:{

PersId: 1;

first_name: “Alvaro”,

surname: “Ortega”,

city: “Valencia”}

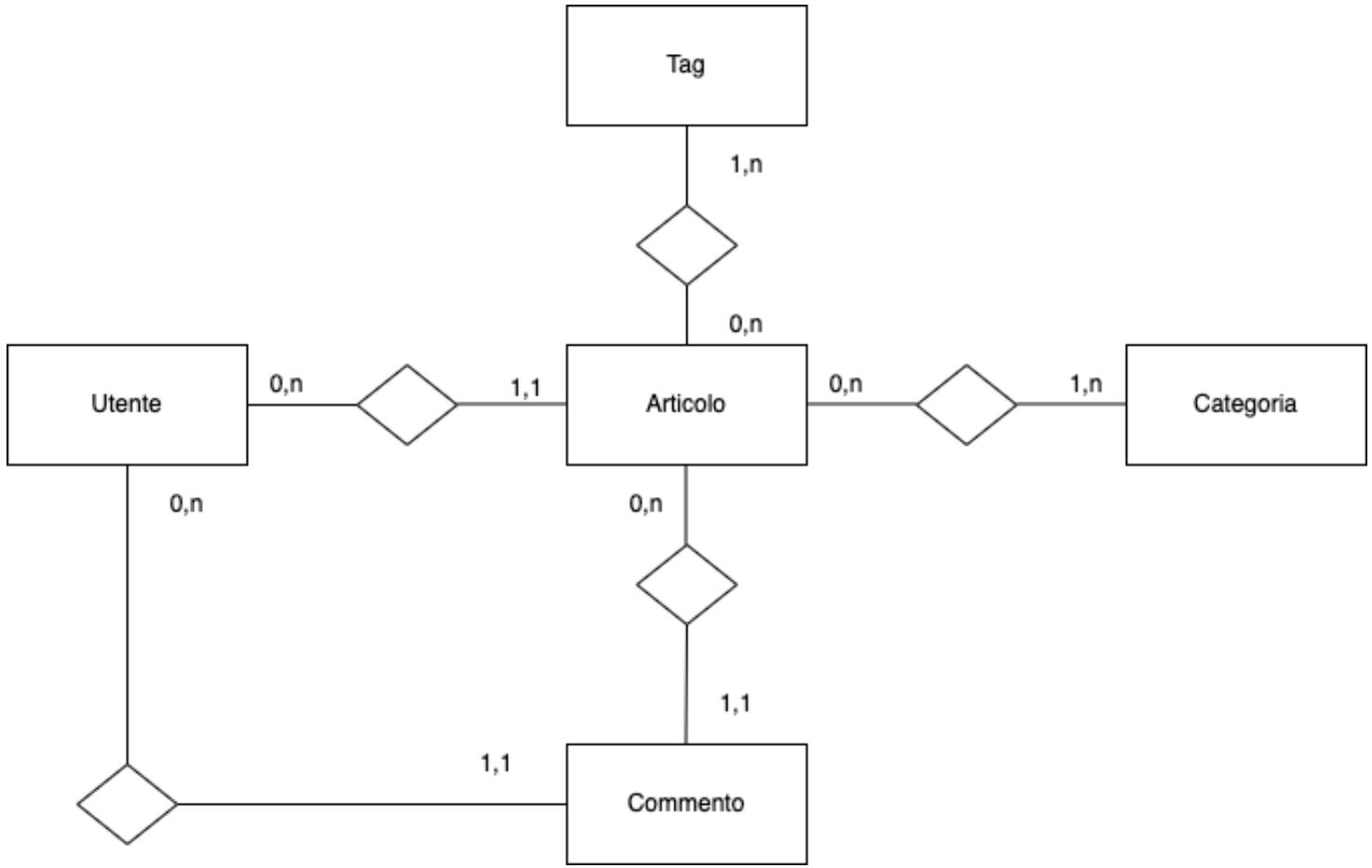
}

Se Alvaro è proprietario di più automobili, i dati ad esso relativi sono replicati tante volte per quante sono le auto di cui è proprietario.

In entrambe le rappresentazioni, gli oggetti tra loro associati sono incorporati (*embedded*) in un unico documento.

La scelta di quale delle due rappresentazioni utilizzare dipende dal tipo di interrogazioni che vengono fatte sulla BD. Se l’accesso è per *persona*, si preferisce la prima rappresentazione, se è per *auto*, la seconda.

Rappresentazione di associazioni tramite referencing



Rappresentazione di associazioni tramite referencing

- **Document “utente” {**
  codice: 101;
  nome: “Paolo”
  email: paolo@ppp.it
  articoli: [
  {codice: xyz
  titolo: “guerra in ucraina”
  data: 3/3/2022
  testo: “abc def ….”
  url: http ://myblog.com
  categorie: [{politica,…}, {attualità, …}, ….]
  tag: [{tagA, ….}, {tagB,…}, …]
  commenti:
  [{date: 3/322021; text: “abc”; utente: <101>},
  {date: 3/4/2022; text: “xyz”, utente: <105>}]
  }

Soluzione A. Rappresentazione per utente – conveniente quando si accede prevalentemente per utente

- In questo documento gli articoli di cui l’utente 101 è autore sono embedded
- Ogni articolo incorpora (1) un array di oggetti che rappresentano le categorie, (2) un array di oggetti che rappresentano i tag, e (3) un array di oggetti che rappresentano i commenti
- Ad ogni commento è associato un utente per *referencing*

Rappresentazione di associazioni tramite referencing

Document “articolo” {
codice: xyz
titolo: “guerra in ucraina”
data: 3/3/2022
testo: “la guerra è un orrore”
url: http ://myblog.com
utente: {
identifier: 101
nome: “Paolo”
email: paolo@ppp.it }
categorie: [{politica,…}, {attualità, …}, ….];
tag: [{tagA, ….}, {tagB,…}, …];
commenti:
[{date: 3/3/2021; text: “abc”, utente: <101>},
{date: 3/4/2022; text: “xyz”, utente: <105>}]

Soluzione B. Rappresentazione per articolo – conveniente quando si accede prevalentemente per articolo

- la relazione articolo-autore è embedded, mentre quella commento-utente è referenced.
- Questa rappresentazione richiede la duplicazione delle informazioni sugli utenti

Rappresentazione di associazioni tramite referencing

Document utente {

- **codice: 101;**
- **nome: “Paolo”**
- **email: paolo@ppp.it**

}

Soluzione C. Rappresentazione per articolo e utente

- In questo caso, tutte le relazioni sono di tipo *referencing*.

Rappresentazione di associazioni tramite referencing

- **Document articolo {**
  codice: xyz
  titolo: “guerra in ucraina”
  data: 3/3/2022
  testo: “ppp … qqqq”
  url: http ://myblog.com
  utente: <101>
  categorie: [{politica,…}, {attualità, …}, ….]
  tag: [{tagA, ….}, {tagB,…}, …]
  commenti:
  [{date: 3/3/2021; text: “abc”, utente: <102>},
  {date: 3/4/2022; text: “lmn”, utente: <103>}]
  }

Soluzione C. Rappresentazione per articolo e utente

- In questo caso, tutte le relazioni sono di tipo *referencing*.

Rappresentazione di associazioni tramite referencing

Document utente {

codice: 101;

nome: “Paolo”

email: paolo@ppp.it

articoli: [articolo: <xyz>,

articolo: <abc>, ….]

}.

Soluzione D. Una soluzione alternativa alle precedenti è quella in cui ad ogni utente è associato un array di riferimenti agli articoli di cui è autore

Rappresentazione di associazioni tramite referencing

Document utente {

codice: 102;

nome: “Paolo”

email: paolo@ppp.it

articoli: [articolo: <xyz>,

articolo: <ccc>, ….]

}.

Questo utente contiene, tra i suoi articoli, ‘xyz’ di cui anche *l’utente 101* è autore, in tal modo alterando la natura dell’associazione (da uno-a-molti a molti-a-molti).

Ciò è coerente col fatto che un Document DB, a differenza di un DB relazionale, non deve essere conforme ad uno schema predefinito.

Embedding vs Referencing

- Il vantaggio dell'embedding è quello di concentrare in un unico documento pezzi di informazione correlate che sono usate contestualmente.
- In tal modo è sufficiente un unico accesso per catturare tutta l'informazione cercata, piuttosto che fare più operazioni di join per ricostruirla.
- Ciò ha un notevole impatto positivo sull'efficienza delle interrogazioni.

Embedding vs Referencing

- Vi sono anche svantaggi, legati al fatto che in genere l'embedding comporta duplicazione dell'informazione
- La ridondanza dei dati ha un impatto negativo sulle operazioni di aggiornamento – la modifica di un proprietario deve essere replicata più volte
- Il referencing evita i problemi di ridondanza e consistenza dei dati ma, al contempo, ha un impatto negativo sulle prestazioni delle interrogazioni a causa della necessità di effettuare operazioni di join.
- In genere, per le associazioni uno-a-uno e uno-a-molti si usa l'embedding, per le associazioni molti-a-molti il referencing

Flessibilità

Document utente {

_id: <utenteId3>

codice: 103;

nome: “Maria”

email: maria@ppp.it;

città: “Roma”;

articoli: [articolo_id: <artId5>,

articolo_id: <artId99>, ….]

}

- A differenza dei database relazionali, dove è necessario dichiarare lo schema di una tabella prima di inserire i dati, le collezioni non richiedono che i loro documenti debbano aderire ad un unico schema
- Cioè, i documenti di una collezione non devono necessariamente avere lo stesso insieme di attributi

Interrogazione

```
db.automobile.insertOne(
    {car_id: “101”,
    model: “Bentley”,
    year: “1973”,
    value: “100.000”,
    owner:{
    PersId: “1”,
    first_name: “Alvaro”,
    surname: “Ortega”,
    city: “Valencia”} )
```

- Non esiste uno standard comune, a differenza dei sistemi relazionali, che aderiscono allo standard SQL come linguaggio di definizione, interrogazione e manipolazione dei dati

Interrogazione

db.automobile.insertOne(

{car_id: “101”,

model: “Bentley”,

year: “1973”,

value: “100.000”,

owner:{

PersId: “1”,

first_name: “Alvaro”,

surname: “Ortega”,

city: “Valencia”})

Identificativo delle auto di marca Bentely costruite prima del 1980

- **db.automobile.find(**
- **{model: “Bentley”, year:{$lt 1980}} // *selezione***
- **{car_id}) // *proiezione***

equivalente a

- **Select car_id**
- **From Automobile**
- **Where model=Bentley and year < 1980**

Interrogazione

db.automobile.insertOne(

{car_id: “101”,

model: “Bentley”,

year: “1973”,

value: “100.000”,

owner:{

PersId: “1”,

first_name: “Alvaro”,

surname: “Ortega”,

city: “Valencia”})

Identificativo e modello delle auto di cui è proprietario Alvaro di Valencia

- **db.automobile.find(**
- **{owner: {first_name: “Alvaro”, city:”Valencia”}}**
- **{car_id, model})**

equivalente a

- **Select car_id, model**
- **From automobile as A, persona as P**
- **Where A.owner=P.persId and P.first_name=alvaro and P.city=Valencia**

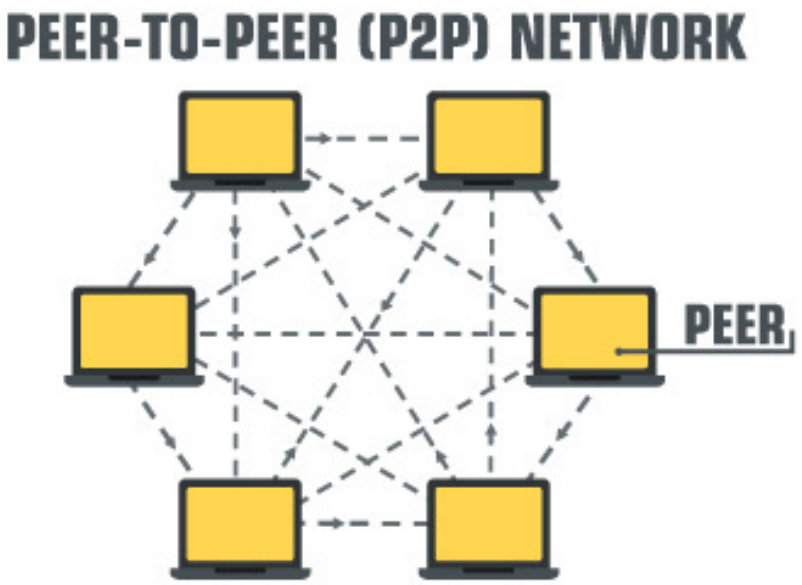
Architettura distribuita

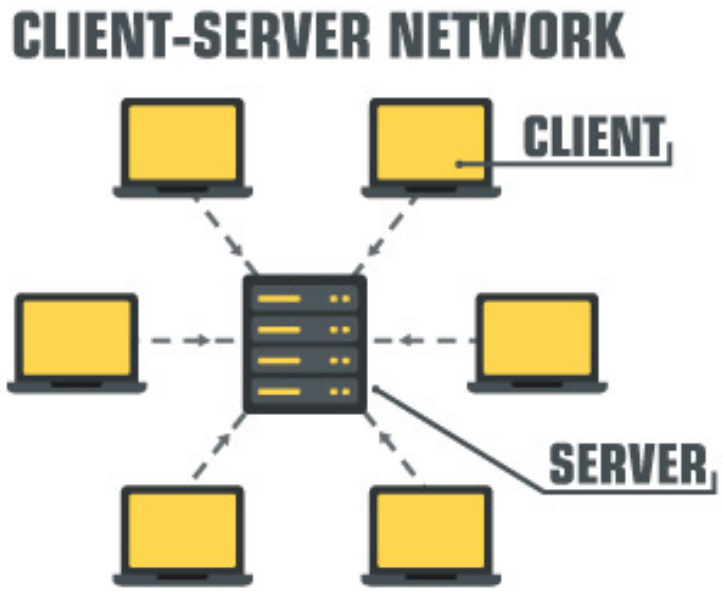
- *Peer-to-Peer* (P2P)
  - ogni nodo agisce come peer e ha gli stessi diritti e responsabilità degli altri nodi. Non esiste un nodo centrale o un coordinatore designato. Ogni nodo conserva una parte dei dati e può interagire direttamente con altri nodi per scambiare informazioni o effettuare operazioni.
  - In un sistema P2P, ogni nodo è equipaggiato con il sw che consente ai nodi di coordinarsi tra loro per il trasferimento dei dati, la propagazione delle modifiche e la gestione della coerenza.

Architettura distribuita

- *Architettura centralizzata (client-server)*
  - Esiste un nodo centrale (coordinatore) che gestisce le operazioni di coordinamento e la distribuzione dei dati tra i nodi
  - Il nodo centrale è responsabile dell'allocazione dei dati, del routing delle query e dell'elaborazione delle transazioni

Architettura distribuita





Architettura distribuita - P2P vs client-server

- In generale, una architettura distribuita consente maggiore efficienza, scalabilità orizzontale e maggiore sicurezza
- In un'architettura centralizzata, il nodo centrale rappresenta tuttavia un singolo punto di fallimento e può limitare la scalabilità orizzontale, in quanto tutti i dati e le operazioni devono passare attraverso il nodo centrale

Distribuzione dei dati

- Lo *sharding* consiste nel suddividere i dati e distribuirli su più nodi
- Ogni nodo è responsabile solo di una parte dei dati, consentendo una maggiore capacità di archiviazione e di elaborazione parallela
- Una strategia comune è utilizzare una funzione hash per assegnare ogni dato a un nodo in base al suo valore hash. In questo modo, i dati con lo stesso valore di hash vengono inviati allo stesso nodo. Ciò consente una distribuzione uniforme dei dati tra i nodi

Distribuzione dei dati

- Consideriamo un'applicazione che gestisce i profili degli utenti con informazioni personali, preferenze e cronologie di attività.
- L'applicazione registra queste informazioni per milioni di utenti e il volume dei dati cresce rapidamente.
- Per gestire questa mole di dati, potremmo utilizzare un database NoSQL che supporta lo sharding
- Supponiamo che abbiamo scelto di suddividere i dati degli utenti in base all'ID utente
- Possiamo utilizzare una funzione hash per determinare a quale shard i dati di un utente devono essere assegnati
- Ad esempio, se abbiamo 4 shard, possiamo utilizzare la funzione hash per assegnare ogni ID utente a uno dei 4 shard disponibili.

Replicazione dei dati

- Nei sistemi NoSQL (Not Only SQL), il concetto di replica dei dati è comune e viene spesso utilizzato per garantire la disponibilità e la ridondanza dei dati.
- La replica dei dati pone problemi legati alla consistenza delle copie aggiornate

Query processing

P2P - una query può essere sottoposta ad un nodo qualsiasi della cluster

- *Query flooding*: una query viene inviata a tutti i nodi del sistema P2P. Ogni nodo esegue la query sui propri dati locali e restituisce i risultati al mittente. Il mittente, quindi, combina e filtra i risultati ricevuti dai nodi per ottenere il risultato finale della query. Questo approccio può generare un elevato traffico di rete e richiedere un'elaborazione aggiuntiva per combinare i risultati
- *Routing delle query*: Quando una query viene sottoposta di un nodo, questo indirizza la richiesta al nodo appropriato. Questo può essere fatto utilizzando la funzione hash usata per lo sharding, o attraverso opportuni indici distribuiti, che consentono una rapida localizzazione dei dati.

Query processing

Client-server - una query è sottoposta al nodo centrale

- *Routing delle query*: Quando una query giunge al nodo centrale, viene analizzata per determinare i dati necessari e le operazioni da eseguire. Il nodo centrale utilizza le informazioni di distribuzione dei dati e l'architettura del database per identificare i nodi specifici che contengono i dati richiesti.
- *Recupero dei dati*: Una volta che i nodi appropriati hanno eseguito le operazioni richieste, restituiscono i risultati al nodo centrale. Il nodo centrale combina e filtra i risultati ricevuti dai nodi per ottenere il risultato finale della query. Questo risultato può essere restituito al mittente della query o a un'applicazione che ha richiesto i dati.

Aggiornamenti

Client-server - un aggiornamento è sottoposto al nodo centrale

1. *Replicazione sincrona*: In questo approccio, il nodo principale aspetta che tutti i nodi replica confermino la ricezione e l'applicazione degli aggiornamenti prima di considerare l'operazione completata. Ciò garantisce che tutte le repliche siano sempre allineate con il nodo principale. Tuttavia, può influire sulle prestazioni complessive del sistema a causa dell'attesa per la conferma delle repliche.
2. *Replicazione asincrona*: In alternativa, la replicazione asincrona consente al nodo principale di inviare gli aggiornamenti ai nodi replica senza aspettare la loro conferma immediata. Il nodo principale considera l'operazione completata una volta che ha inviato gli aggiornamenti ai nodi replica. Ciò può migliorare le prestazioni del sistema, ma potrebbe comportare un certo ritardo nella sincronizzazione delle repliche.

Aggiornamenti

P2P - un aggiornamento può essere sottoposto ad un qualsiasi nodo

- Un diffuso approccio è il *diffusion-based replication,* che prevede che quando un nodo effettua un aggiornamento, lo invia a tutti gli altri nodi nel sistema. Ogni nodo riceve l'aggiornamento e lo applica localmente. Questo approccio può generare un elevato traffico di rete, ma garantisce che tutti i nodi abbiano sempre gli stessi dati
- Tipicamente, si tratta di *replicazione asincrona* in cui, dopo che un nodo ha eseguito un aggiornamento, le modifiche vengono propagate in modo asincrono ai nodi replica, senza cioè attendere la conferma da parte di tutti i nodi prima di considerare l'operazione completata. Di conseguenza, è possibile che i nodi replica abbiano temporaneamente versioni diverse dei dati.
- La consistenza dei dati viene eventualmente raggiunta quando tutte le repliche ricevono le modifiche (*eventual consistency*).

Proprietà BASE

- Le caratteristiche di un sistema NoSql sono riassunte dall'acronimo BASE: **B**asically **A**vailable, **S**oft state, **E**ventual consistency:
- *Basically Available* (fondamentalmente disponibile). Nella eventualità che un guasto interrompa l'accesso a un nodo, il sistema continua comunque ad offrire il servizio.
- *Soft state* (stato leggero). lo stato del sistema può cambiare nel corso del tempo, anche senza intervento dell'utente. Ciò a causa della eventual consistency
- *Eventual consistency* (coerenza finale). I NoSql in genere non garantiscono la consistenza, ma solo la *enventual consistency*. Infatti, l'unico requisito che i database NoSQL hanno per quanto riguarda la coerenza è quello di richiedere che ad un certo punto in futuro i dati convergano verso uno stato coerente.

Proprietà BASE

- Si noti la differenza con i sistemi relazionali, dove vi è l'obbligo di coerenza immediata (proprietà ACID), che vieta l'esecuzione di un'operazione fino al completamento della precedente operazione e fino a quando la base di dati non abbia raggiunto uno stato coerente

Sistemi Commerciali

- MongoDB (client-server)
- CouchDB (P2P)
- OrientDB (client-server)
- DocumentDB (client-server)