I DBMS NoSQL

NOT ONLY SQL

Introduzione

Cosa significa NoSQL

Il termine viene usato per la prima volta nel '98 da Carlo Strozzi

- Indicava un RDBMS open-source che usava un linguaggio diverso da SQL per le interrogazioni

Nel 2009 viene usato da un meetup di San Francisco

- Ospitavano discussioni di progetti open-source ispirati ai nuovi database di Google e Amazon
- Gruppi partecipanti: Voldemort, Cassandra, Dynomite, HBase, Hypertable, CouchDB, MongoDB

Oggi, il termine NoSQL indica dei DBMS (DataBase Management System) in cui il meccanismo di persistenza è diverso dal modello relazionale (usato dagli RDBMS)

- NoSQL = Not Only SQL
- Secondo Strozzi, il termine NoREL sarebbe stato più consono

I punti forti degli RDBMS

Meccanismo delle transazioni

- Garanzia nella gestione della consistenza e degli accessi concorrenti

Integrazione

- Applicazioni diverse possono condividere e riutilizzare le stesse informazioni

Standard

- Il modello relazionale ed il linguaggio SQL sono standard affermati
- Un unico background teorico condiviso da diverse tecnologie

Solidità

- In uso da oltre 40 anni

I punti deboli degli RDBMS

Conflitto di impedenza

- La memorizzazione del dato si basa sul modello relazionale, ma la manipolazione del dato si basa tipicamente sul modello a oggetti
- Tante soluzioni proposte, nessuno standard
  - E.g.: Object Oriented DBMS (OODBMS), Object-Relational DBMS (ORDBMS), Object-Relational Mapping (ORM) frameworks

Difficile scalabilità orizzontale

- I Big Data sono una realtà; un unico server non può gestire tutto
- Distribuire un RDBMS non è una soluzione facile

Consistenza vs efficienza

- Garantire la consistenza dei dati è un must – anche a costo delle performance
- Le applicazioni odierne richiedono letture e scritture con grande frequenza e a bassa latenza

Rigidità dello schema

- Una modifica "a regime" può essere molto costosa

Caratteristiche comuni ai NoSQL

Non solo righe e tabelle

- Varietà di modelli per la gestione e la manipolazione dei dati

Libertà dai join

- Possibilità (in alcuni casi, necessità) di estrarre i dati senza eseguire dei join

Libertà dagli schemi

- Possibilità di caricare e interrogare i dati senza definire degli schemi prefissati (*schemaless* o *soft-schema*)

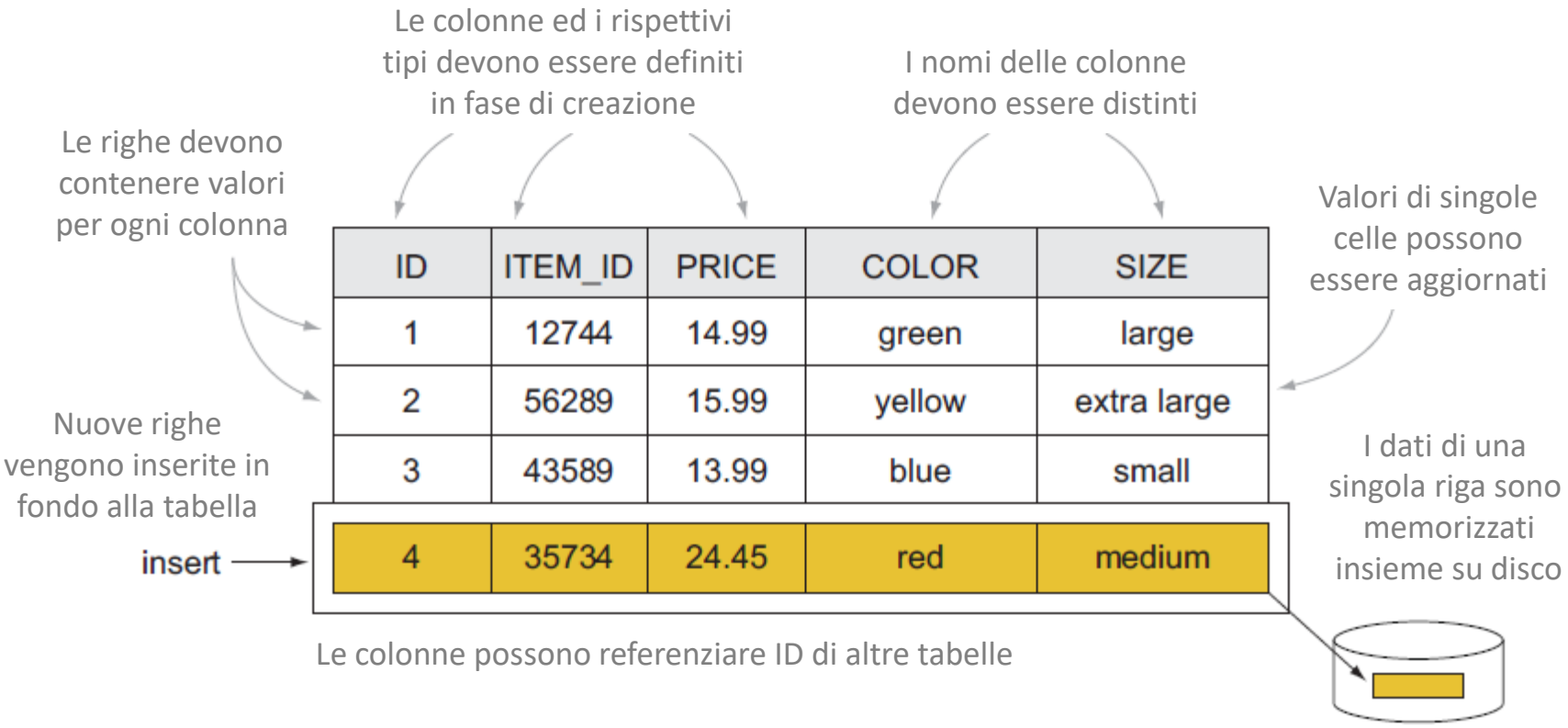
Architettura distribuita e *shared-nothing*

- Semplice scalabilità del sistema in ambiente distribuito senza degrado di performance
- Ogni workstation dispone della propria RAM e dei propri dischi, che non sono condivisi

Modelli dati

Modello relazionale

Il modello relazionale si basa su righe e tabelle



NoSQL: tanti modelli

Una delle principali difficoltà è capire quale modello adottare

Modello	Descrizione	Casi d'uso
Key-value	Associa un qualunque valore ad una stringa di testo	Dizionari, tabelle di lookup, cache, memorizzazione file e immagini
Document	Memorizza informazioni gerarchiche con una struttura ad albero	Documenti, qualunque dato idoneo ad una struttura gerarchica
Column family	Memorizza matrice sparse usando sia la riga che la colonna come chiave	Crawling, sistemi con elevata variabilità, matrici sparse
Graph	Memorizza nodi e archi	Query su reti sociali, inferenza, pattern matching

Key-value: modello

Un DB contiene una o più collezioni (corrispettive delle tabelle)

Ogni collezione contiene un elenco di coppie chiave-valore

- Chiave: una stringa di testo univoca
  - Esempi: id, hash, path, query, chiamate REST
- Valore: un BLOB (binary large object)
  - Esempi: testo, documenti, pagine web, file multimediali

Livello di atomicità: la coppia chiave-valore

Key	Value
image-12345.jpg	Binary image file
http://www.example.com/my-web-page.html	HTML of a web page
N:/folder/subfolder/myfile.pdf	PDF document
9e107d9d372bb6826bd81d3542a419d6	The quick brown fox jumps over the lazy dog
view-person?person-id=12345&format=xml	<Person><id>12345</id .</Person>
SELECT PERSON FROM PEOPLE WHERE PID="12345"	<Person><id>12345</id .</Person>

Ricorda un semplice dizionario

- La collezione è indicizzata per chiave
- Il valore può contenere informazioni diverse: una o più definizioni, sinonimi e contrari, immagini, ecc.

Key-value: interrogazione

Tre semplici modalità di interrogazione:

- put($key as xs:string, $value as item())
  - Aggiunge una coppa chiave-valore alla collezione
  - Se la chiave esiste già, il valore viene rimpiazzato
- get($key as xs:string) as item()
  - Restituisce il valore corrispondente alla chiave indicata (se esiste)
- delete($key as xs:string)
  - Elimina la coppia con la chiave indicata

Il valore è come una *black box*: non lo si può interrogare!

- Non è possibile fare query con filtri
- Non è possibile indicizzare i valori
- Per questo motivo, le chiavi sono spesso "parlanti"

Key	Value
user:1234:name	Enrico
user:1234:age	30
post:9876:written-by	user:1234
post:9876:title	NoSQL Databases
comment:5050:reply-to	post:9876

Document: modello

Un DB contiene una o più collezioni (corrispettive delle tabelle)

Ogni collezione contiene un elenco di documenti (tipicamente JSON)

- Un documento ha una struttura ad albero auto-descrittiva

Ogni documento contiene un insieme di campi

- L'unico campo obbligatorio è l'ID,
  il cui valore identifica il documento nella collezione

Ogni campo è strutturato come una coppia chiave-valore

- Chiave: stringa di testo univoca all'interno del documento
- Valore: può essere semplice (stringa, numero, booleano)
  o complesso (oggetto, array, BLOB)
  - Un valore può contenere campi a sua volta

Livello di atomicità: il documento

```
{
    "_id": 1234,
    "name": "Enrico",
    "age": 30,
    "address": {
        "city": "Ravenna",
        "postalCode": 48124
    },
    "contacts": [ {
        "type": "office",
        "contact": "0547-338835"
    }, {
        "type": "skype",
        "contact": "egallinucci"
    } ]
}
```

Document: interrogazione

A differenza dei key-value, il contenuto del documento è visibile al DBMS

Di conseguenza, l'espressività dei linguaggi di interrogazione è molto più elevata

- Possibile creare indici sui campi
- Possibile filtrare sui campi
- Possibile ottenere più documenti con un'unica query
- Possibile effettuare proiezioni sui documenti
- Possibile eseguire degli update sui campi di un documento

Implementazioni diverse offrono funzionalità diverse

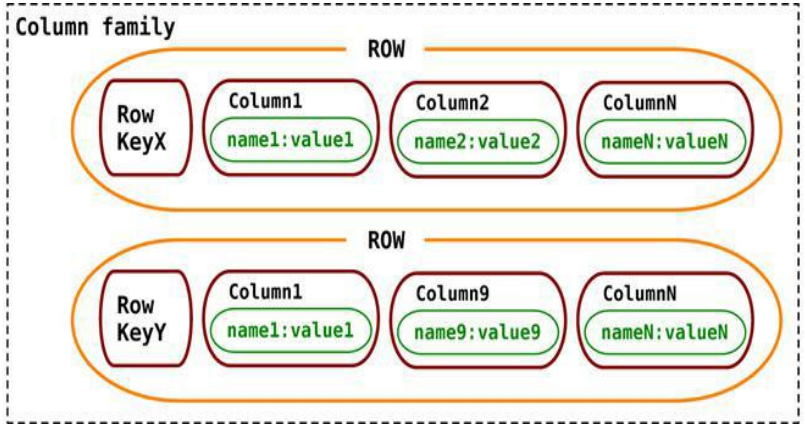
- Meccanismi per gestire viste materializzate o dinamiche
- Interrogazione dei dati secondo il paradigma MapReduce
- Connettori con diversi strumenti Big Data (e.g., Spark, Hive)
- Esecuzione di interrogazioni *full-text*

Column-family: modello

Un DB contiene una o più famiglie di colonne (o tabelle)

Ogni famiglia di colonne contiene un elenco di righe nella forma di coppie chiave-valore

- Chiave: stringa di testo univoca all'interno della famiglia di colonne
- Valore: un insieme di colonne



Ogni colonna è, a sua volta, una coppia chiave-valore

- Chiave: stringa di testo univoca all'interno della riga
- Valore: semplice o complesso (*supercolonna*)

Livello di atomicità: la riga

Rispetto al modello relazionale:

- Le righe specificano solo le colonne per cui esiste un valore
  - Soluzione particolarmente adeguata per matrici sparse
- Il valore di una colonna può essere "versionato" sulla base di un timestamp

Famiglia di colonne	**Chiave di riga**	**Chiave di colonna**	**Timestamp**	**Valore**

Column-family : interrogazione

L'espressività dei linguaggi è una via di mezzo tra i key-value ed i document

- Sconsigliato (ove possibile) creare indici sulle colonne
- Possibile filtrare sulle colonne (con alcune limitazioni)
- Possibile ottenere più righe con un'unica query
- Possibile effettuare proiezioni sulle righe
- Impossibile eseguire degli update sulle colonne di un documento
  - Possibile solo rimpiazzarla interamente, come nei key-value

Data la similarità col modello relazionale, è spesso disponibile un linguaggio SQL-like

Graph: modello

Un DB contiene uno o più grafi

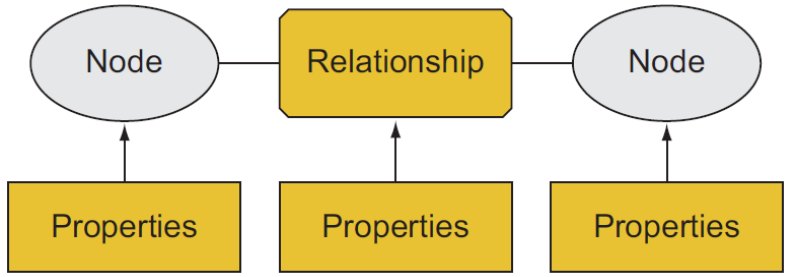
Ogni grafo è composto da nodi e relazioni

- Nodi: rappresentano solitamente entità reali
  - E.g.: persone, organizzazioni, pagine web, workstation, cellule, libri, ecc.
- Relazioni: rappresentano connessioni direzionate tra i nodi
  - E.g.: amicizie, relazioni di lavoro, collegamenti ipertestuali, collegamenti ethernet, diritti d'autore, ecc.
- Nodi e relazioni sono descritti da proprietà

Livello di atomicità: la transazione

Specializzazioni più conosciute:

- Modello reticolare
  - Principalmente relazioni di tipo parent-child o owner-member
- Triplestore
  - Relazioni soggetto-predicato-oggetto (e.g., RDF)



Graph: interrogazione

I database a grafo modellano situazioni completamente diverse

Di conseguenza, anche linguaggi e modalità di interrogazione cambiano

- Supporto alle transazioni
- Supporto a indici, selezioni e proiezioni
- Linguaggio di interrogazione basato su pattern

Query	Pattern
Trova amici di amici	(user)-[:KNOWS]-(friend)-[:KNOWS]-(foaf)
Trova il percorso più breve tra A e B	shortestPath((userA)-[:KNOWS*..5]-(userB))
Cosa ha comprato chi ha comprato i miei stessi prodotti?	(user)-[:PURCHASED]->(product)<-[:PURCHASED]-()-[:PURCHASED]->(otherProduct)

Gestione della consistenza

UNO SGUARDO DIETRO LE QUINTE

RDBMS vs NoSQL: filosofie diverse

I DBMS relazionali vengono da decenni di utilizzo assodato

- Forte focus sulla consistenza dei dati
- Ottimizzazione delle performance maturata con anni di ricerca
- Sistemi altamente complessi (trigger, caching, sicurezza, ecc.)

I sistemi NoSQL nascono per far fronte alle limitazioni degli RDBMS

- Forte focus sulla distribuzione e sulla disponibilità dei dati
- Sistemi molto semplici (per ora)
- Rapidità e maneggevolezza piuttosto che consistenza a tutti i costi

Consistenza: un esempio

Immaginiamo di fare un giroconto di 1000€ dal conto A al conto B; il passaggio richiede due operazioni

- Prelievo di 1000€ dal conto A
- Deposito di 1000€ sul conto B

In nessun caso deve poter capitare che:

- A causa di un errore, i 1000€ non vengano depositati sul conto B e vengano persi
- A causa di un errore, i 1000€ vengono depositati due volte sul conto B (magari!)
- Interrogando il database, venga visualizzato uno stato intermedio
  - E.g., se in A c'erano solo 1000€ e in B 0€, non deve poter essere visualizzato uno stato intermedio in cui A+B = 0€

Negli RDBMS esiste un meccanismo infallibile: le transazioni

Consistenza negli RDBMS: ACID

Le transazioni garantiscono quattro proprietà fondamentali, denominate ACID

Atomicity
- La transazione è indivisibile: o viene completata interamente con successo, o fallisce
- Non è possibile che venga completata solo a metà

Consistency
- La transazione lascia il DB in uno stato consistente
- Nessuno dei vincoli di integrità del DB può mai essere violato

Isolation
- Una transazione esegue indipendentemente dalle altre
- Se più transazioni eseguono in concorrenza, l'effetto netto equivale a quello di un'esecuzione seriale

Durability
- Il DBMS protegge il DB a fronte di guasti

Consistenza negli RDBMS: ACID

L'implementazione delle transazioni ACID richiede la gestione di un meccanismo di lock e log

- Blocco delle risorse e tracciamento delle modifiche
- In caso di problemi, ripristino dello stato iniziale
- Alla fine, sblocco delle risorse

Garanzia di consistenza a discapito di velocità e disponibilità del dato

- Ricadute sui tempi di attesa degli utenti
- Difficili da gestire se il database è distribuito

In alcune situazioni, la garanzia di consistenza è meno importante

- E.g.: acquisto di prodotti
- La gestione del carrello richiede velocità e disponibilità
- L'emissione dell'ordine richiede consistenza

Consistenza nei NoSQL

Diversi tentativi sono stati fatti per descrivere le proprietà dei sistemi NoSQL in contrapposizione alle proprietà ACID delle transazioni

- Proprietà BASE
- Teorema CAP
- Teorema PACELC

Non vanno intese come proprietà garantite dai sistemi NoSQL…

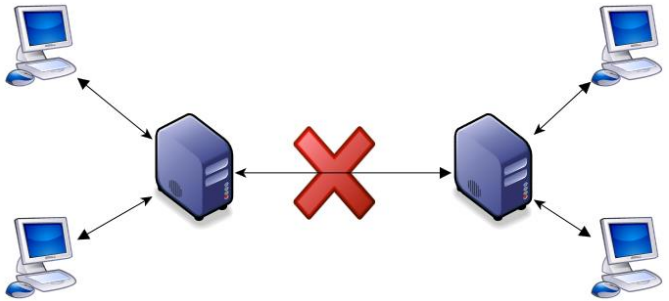
… ma come proprietà che *tentano* di descriverne il comportamento

Consistenza nei NoSQL: esempio

Anna vuole prenotare l'ultima stanza disponibile all'hotel Hilton di New York da un nodo che si trova a Londra

Bob vuole fare lo stesso da un nodo che si trova a Mumbai

Avviene un partizionamento di rete



Consistenza nei NoSQL: BASE

Basic **A**vailability
- Il sistema deve essere sempre disponibile

Soft-state
- E' accettabile che il sistema presenti delle inconsistenze, purché temporanee

Eventual consistency
- Prima o poi, il sistema verrà lasciato in uno stato consistente
- Il concetto più importante

Prof. Eric. A. Brewer
(Berkeley, Google)

ACID
- Approccio pessimista (prevenire meglio che curare)

BASE
- Approccio ottimista (prima o poi si sistema tutto)
- Focus su throughput piuttosto che su consistenza

Consistenza nei NoSQL: teorema CAP

"Teorema": date le seguenti tre proprietà, solo due possono essere garantite

Consistency: il sistema è sempre consistente

Availability: il sistema è sempre disponibile

Partition tolerance: il sistema tollera i partizionamenti di rete

Prof. Eric. A. Brewer
(Berkeley, Google)

Tre situazioni

- CA: il sistema non può subire partizionamenti (single server)
- AP: in caso di partizionamenti, il sistema sacrifica la consistenza (overbooking)
- CP: in caso di partizionamenti, il sistema sacrifica la disponibilità (prenotazioni bloccate)

Teorema controverso e di difficile interpretazione

- Asimmetria delle proprietà
- Cosa significa non tollerare partizionamenti?
- CA e CP sono la stessa cosa?

Consistenza nei NoSQL: teorema PACELC

"Teorema": evoluzione del teorema CAP
- if (**P**artition) then { **A**vailability or **C**onsistency? }
- **e**lse { **L**atency or **C**onsistency }

Distingue il comportamento in presenza di partizionamenti da quello "normale"
- PA: in caso di partizionamenti, il sistema sacrifica la consistenza (overbooking)
- PC: in caso di partizionamenti, il sistema sacrifica la disponibilità (prenotazioni bloccate)
- EL: a regime normale, il sistema sacrifica la consistenza in favore della latenza
- EC: a regime normale, il sistema sacrifica la latenza in favore della consistenza

Quattro situazioni:
- PA EL: sistema incentrato su velocità e disponibilità (principale filosofia dei sistemi NoSQL)
- PA EC: consistenza sacrificata solo in caso di partizionamento
- PC EL: consistenza sacrificata solo a regime normale (e.g., Yahoo Sherpa)
- PC EC: sistema fortemente consistente (RDBMS)

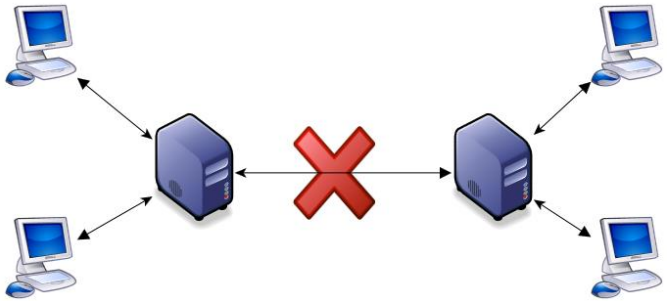
Prof. Daniel Abadi (Maryland)

Consistenza nei NoSQL: esempio

Anna vuole prenotare l'ultima stanza disponibile all'hotel Hilton di New York da un nodo che si trova a Londra

Bob vuole fare lo stesso da un nodo che si trova a Mumbai

Avviene un partizionamento di rete



Consistenza nei NoSQL: soluzioni

CP: nessun utente può prenotare stanze (si sacrifica A)
- Situazione non ideale

AP: entrambi i nodi accettano le richieste di prenotazione (si sacrifica C)
- Possibile overbooking: conflitto in scrittura da gestire

caP: il nodo di Mumbai viene designato come master per l'hotel Hilton
- Solo Bob può fare la prenotazione
- Anna vede l'informazione (inconsistente) e non può fare la prenotazione

La gestione di queste situazioni dipende strettamente dal contesto
- Trading finanziario? Blog? Acquisti online?

L'importante è capire:
- Qual è la tolleranza sulla lettura di dati obsoleti
- Quanto può essere ampia la finestra di inconsistenza

Gestione della distribuzione dei dati

UNO SGUARDO DIETRO LE QUINTE

Distribuzione dei dati

Uno dei punti di forza dei sistemi NoSQL è la capacità di *scalare*

- L'aggregato è un'unità che si presta bene alla distribuzione all'interno di un cluster

In generale, gestire un cluster introduce una certa complessità

- L'utilizzo di un database NoSQL in un server singolo non è da escludere

In un sistema distribuito ci sono due aspetti ortogonali da valutare:

- **Sharding**: spacchettare i dati su nodi diversi
- **Replication**: copiare gli stessi dati su nodi diversi
  - Modalità master-slave
  - Modalità peer-to-peer

Single-server

La soluzione più semplice: utilizzare il database in un'unica macchina

- Molti database NoSQL progettati per lavorare in forma distribuita..
- ..ma non significa che non lavorino bene in locale
- I database a grafo lavorano meglio in locale

Quando ha senso l'approccio single-server?

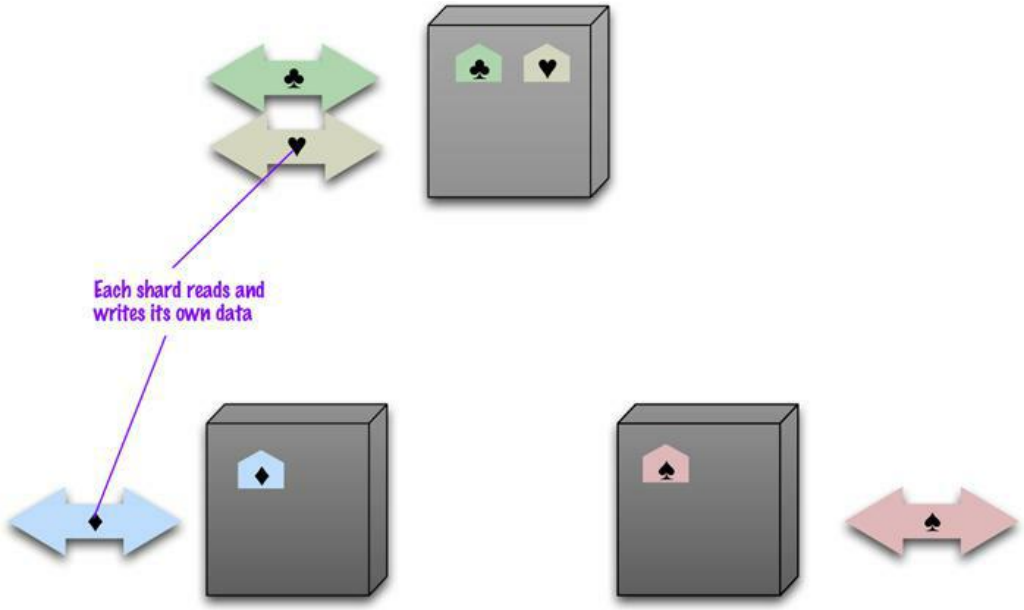
- Mole di dati non è enorme
- Si vogliono sfruttare le caratteristiche dei database NoSQL (e.g., schemaless, modello dati)

Un sistema distribuito è inevitabilmente più complesso; non è detto che ne valga sempre la pena

Sharding

Sharding: spezzare i dati in parti (*shard*) che vengono memorizzate su macchine diverse

- I.e., scalare orizzontalmente



Una buona *strategia di sharding* è **fondamentale** per ottimizzare le performance

- Tipicamente basata sul valore di uno o più campi

Strategie di sharding

Regole principali per una strategia di sharding:

1. Memorizzare i dati vicini al luogo da cui devono essere acceduti
   - E.g., memorizzare gli ordini per l'Italia nel data center europeo
2. Mantenere una distribuzione bilanciata
   - Ogni nodo dovrebbe ricevere (più o meno) la stessa percentuale di dati
3. Mettere insieme dati che potrebbero essere letti in sequenza
   - Stesso cliente, stesso nodo

Replication

Replication: i dati vengono copiati su diversi nodi

Come gestire la distribuzione delle repliche nei nodi?

- Principio comunemente adottato:
  - Una replica nel nodo che riceve il dato
  - Una replica in un nodo diverso all'interno dello stesso rack
  - Una replica in un nodo all'interno di un rack diverso

Ogni aggiornamento ad un dato richiede la propagazione delle modifiche a tutte le sue repliche

Due tecniche per gestire gli aggiornamenti:

- Master-slave
- Peer to peer

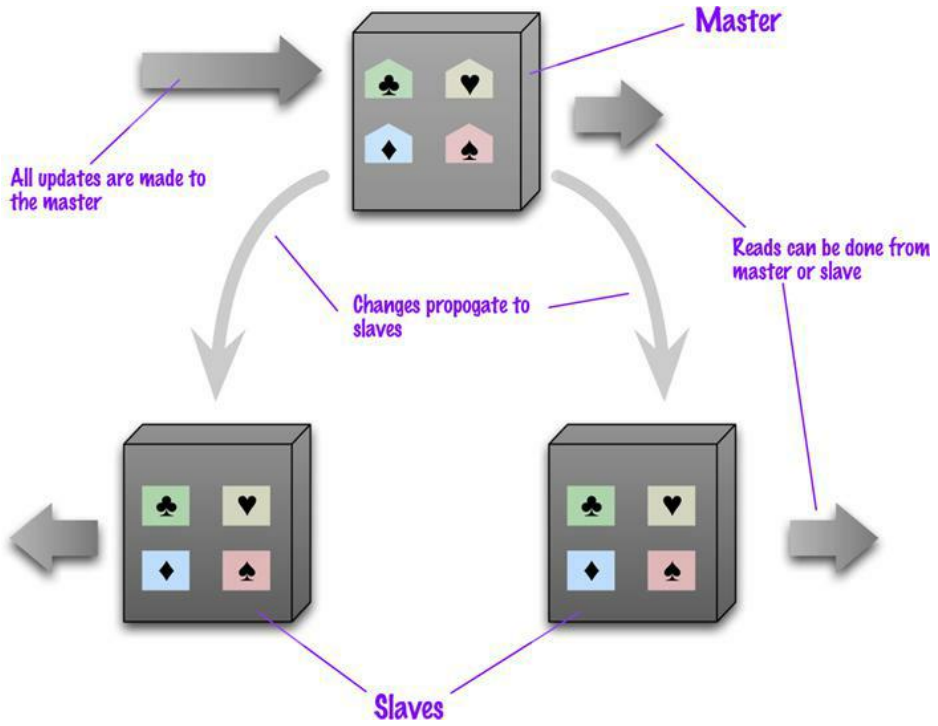
Master-Slave Replication

Master

- E’ il "manager" dei dati
- Gestisce tutti i processi di aggiornamento
- Può essere deciso manualmente o sorteggiato

Slaves

- Consentono la lettura dei dati
- Sincronizzati con il master
- Possono diventare master in caso di failure di quest’ultimo



Master-Slave Replication: pro e contro

Pro

- Gestisce facilmente molte richieste di lettura
  - Basta aggiungere più nodi e assicurarsi che le letture vengono gestite degli slave
- Gli slave possono gestire le letture anche senza master
- Utile quando il carico di lavoro è principalmente in lettura

Contro

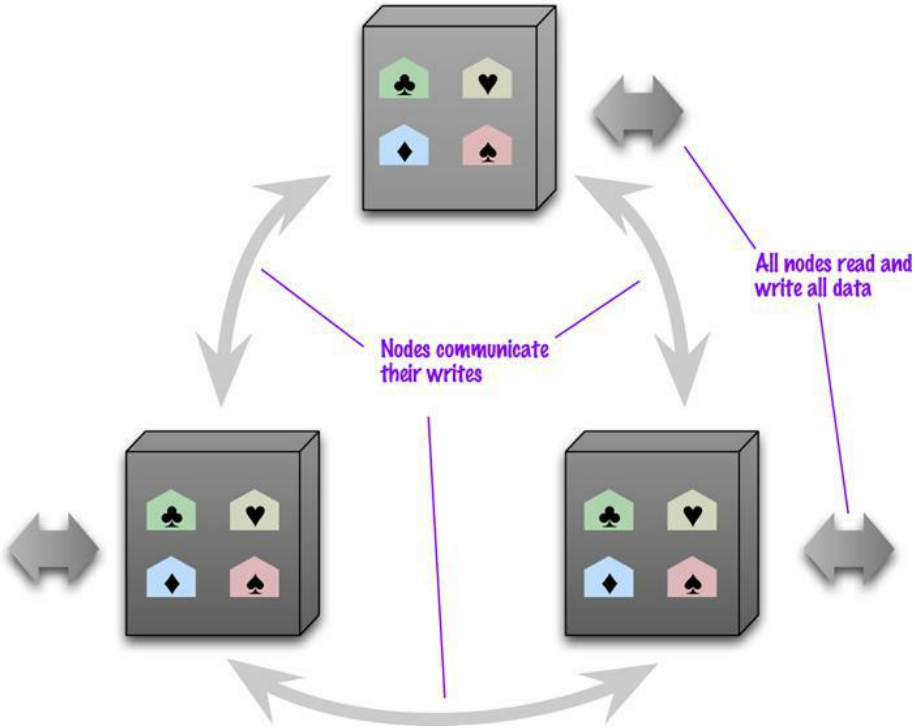
- Il master è un collo di bottiglia
  - **Solo il master può gestire gli aggiornamenti dei dati**
  - In caso di failure, bisogna attendere il ripristino del master o una nuova elezione
- La lenta propagazione dei cambiamenti può generare inconsistenza
- Non ideale in presenza di un forte carico di lavoro in scrittura

Peer-to-Peer Replication

Tutti i nodi hanno la stessa importanza

Tutti i nodi gestiscono gli aggiornamenti

La perdita di un nodo non compromette né letture né scritture



Peer-to-peer Replication: pro e contro

Pro

- Il failure di un nodo non interrompe le richieste di letture e di scritture
- Aggiungendo nodi di aumentano facilmente le performance

Contro

- Inconsistenza!
- La propagazione dei cambiamenti da un nodo all'altro è lenta
  - Le inconsistenze in lettura sono problematiche, ma relativamente brevi
- Due persone possono aggiornare diverse copie dello stesso dato (memorizzate in nodi diversi) nello stesso momento
  - Le inconsistenze in scrittura sono per sempre

One size does not fit all

AD OGNI APPLICAZIONE IL GIUSTO MODELLO DATI

DB Key-Value popolari

Riak: http://basho.com/riak/

Redis (Data Structure server): http://redis.io/

- Più di un semplice key-value perché supporta la memorizzazione di strutture più complesse (liste, set, …) e l'esecuzione di operazioni sui valori (range, diff, …)

Memcached DB: http://memcached.org/

Berkeley DB: http://www.oracle.com/us/products/database/berkeley-db/

HamsterDB: http://hamsterdb.com/

- Specialmente per usi embedded

Amazon DynamoDB: https://aws.amazon.com/dynamodb/

- Un servizio non open-source

Project Voldemort: http://www.project-voldemort.com/

- Un'implementazione open-source di Amazon DynamoDB

DB Key-Value: quando usarli

Memorizzare informazioni di sessione

- Ogni sessione web è unica e ha un valore univoco di sessionId. Tutte le informazioni su una sessione possono essere memorizzare con una richiesta PUT e recuperate con una richiesta GET.

Profili utente, preferenze

- Ogni utente ha il suo identificatore (userId, username, o altro) e ha le sue preferenze in termini di lingua, colori, timezone, elenco dei prodotti accessibili, ecc. – tutte informazioni che possono essere messe in un unico aggregato e recuperate con un semplice GET.

Shopping Cart, servizi di chat

- Ogni sito di e-commerce associa un carrello ad ogni utente. Dato che l'accesso a questi carrelli deve essere sempre garantito, indipendentemente dal browser, dalla macchina o dalla sessione dell'utente, tutte le informazioni sullo shopping dell'utente possono essere memorizzate in un aggregato la cui chiave è l'ID dell'utente.

DB Key-Value: quando non usarli

Dati con relazioni

- Se c'è bisogno di gestire le relazioni fra dati in dataset diversi (o fra dati con set di chiavi diversi) – anche se alcuni di essi forniscono meccanismi di link-walking.

Transazioni multi-operazione

- Se si vuole la garanzia che un'operazione che coinvolge più chiavi avvenga in maniera atomica (e non in maniera parziale).

Interrogazioni sui dati

- Se si vogliono effettuare interrogazioni sulle informazioni memorizzate nel valore dell'aggregato anziché sulla chiave – pochi prodotti consentono di farlo (e.g., Riak Search), ma con funzionalità limitate.

Operazioni su insiemi di dati

- Dato che le operazioni sono limitate ad una chiave alla volta, non c'è modo di operare contemporaneamente su più chiavi – l'unico modo per farlo è lato applicazione.

DB documentali popolari

MongoDB: http://www.mongodb.org

CouchDB: http://couchdb.apache.org

Terrastore: https://code.google.com/p/terrastore

OrientDB: http://www.orientechnologies.com

RavenDB: http://ravendb.net

Lotus Notes: http://www-03.ibm.com/software/products

DB documentali: quando usarli

Log di eventi / web services

- I DB documentali possono essere utilizzati come repository centrali per la memorizzazioni di log di eventi di diverse applicazioni. Una buona pratica può essere quella di attivare lo sharding sulla base del nome dell'applicazione sorgente o del tipo di evento

CMS, piattaforme di blogging

- L'assenza di uno schema predefinito rende i database documentali adatti all'utilizzo per content management systems (CMS) o per applicazioni di gestione di siti web, per la gestione di commenti, registrazioni e profili utente

Web Analytics o Real-Time Analytics

- La possibilità di aggiornare solo alcune parti di un documento permette di salvare e aggiornare facilmente delle metriche di analisi. L'indicizzazione di contenuti testuali abilita anche applicazioni di real-time sentiment analysis e social media monitoring.

Applicazioni di e-commerce

- Le applicazioni di e-commerce hanno spesso bisogno di flessibilità sullo schema per memorizzare prodotti e ordini e per poter permettere l'evoluzione del proprio modello dati senza incorrere in costi di refactory o di migrazione dei dati

DB documentali: quando non usarli

Transazioni complesse e multi-operazione

- A meno di qualche eccezione (e.g., RavenDB), i database documentali non sono adatti per operazioni atomiche cross-documento.

Interrogazioni su strutture aggregate variabili

- Se la struttura degli aggregati cambia continuamente, anche le query devono essere costantemente modificate. Una soluzione potrebbe essere quella di disaggregare i dati (i.e., normalizzarli), ma verrebbe meno il concetto di aggregato.

DB column-family popolari

Cassandra: http://cassandra.apache.org

HBase: https://hbase.apache.org

Hypertable: http://hypertable.org

SimpleDB: https://aws.amazon.com/simpledb

Google BigTable: https://cloud.google.com/bigtable

DB column-family: quando usarli

Log di eventi; CMS, piattaforme di blogging

- Un sistema di logging può trarre beneficio dalla facilità di scalare rispetto alla scritture e dalla possibilità di utilizzare colonne diverse a seconda dell'applicazione. Una buona strategia può essere quelle di utilizzare **chiavi «parlanti»** nella forma appname:timestamp.

Matrici sparse

- Mentre un RDBMS gestisce tutti i *null*, un database colonnare permette di allocare solamente le colonne per cui è necessario memorizzare un dato.

Applicazioni GIS

- La possibilità di gestire un numero eccezionale di colonne si presta bene ad applicazioni GIS, ad esempio localizzando "pezzi" di mappa come coppie di longitudine (chiave di riga) e latitudine (colonna)

Contatori

- Spesso, nelle applicazioni web vengono contati e categorizzati i visitatori delle pagine a scopo di analisi. Cassandra offre un tipo di colonna specificamente progettata per la gestione di contatori (Counter).

Gestione della scadenza

- Grazie ai timestamp è possibile impostare delle colonne a scadenza con un determinato Time To Live (TTL): accessi temporanei a fini dimostrativi, annunci pubblicitari per un periodo limitato.

DB column-family: quando non usarli

Sistemi che richiedono transazioni ACID per letture e scritture

Le aggregazioni dei dati per ottenere misure aggregate (e.g., somma, media) **non sono supportate**, ma devono essere gestite lato applicazione

- Esistono tuttavia software (e.g., Apache Hive) che lavorano «sopra» ai database colonnari e che offrono un valido supporto per queste operazioni

DB a grafo popolari

Neo4J: http://neo4j.com

InfiniteGraph: http://www.objectivity.com/products/infinitegraph/

OrientDB: http://orientdb.com/orientdb/

FlockDB: http://github.com/twitter-archive/flockdb

DB a grafo: quando usarli

Dati interconnessi

- I social network sono un'applicazioni in un cui i database a grafo sono molto efficienti (non solo per memorizzare amicizie, ma, per esempio, per memorizzare relazioni di lavoro). Ogni dominio ricco di relazioni è un buon contesto.

Servizi di instradamento e location-based

- Applicazioni che affrontano il problema del commesso viaggiatore (TSP, Travelling Salesman Problem). Applicazioni location-based per suggerire, ad esempio, i migliori ristoranti nelle vicinanze. In questi casi, le relazioni modellano il concetto di distanza tra i nodi

Applicazioni di recommendation, fraud-detection

- Sistemi che suggeriscono «i prodotti acquistati dai tuoi amici», o «i prodotti acquistati da chi ha acquistato questo prodotto». Quando le relazioni sono talmente tante da evidenziare chiari pattern comportamentali, la ricerca di outlier può servire per la rilevazione di frodi.

DB a grafo: quando non usarli

Applicazioni data-intensive

- Attraversare il grafo è semplice, ma analizzare tutto il grafo può diventare particolarmente oneroso. Esistono framework per l'elaborazione distribuita di grafi (e.g., Apache Giraph), ma non si appoggiano su database a grafo
- L'aggiornamento in cascata di molti nodi (e.g., modificare tutti i nodi che rispondono a determinate caratteristiche) non è un'operazione semplice

La scelta del database

Le due ragioni principali per fare affidamento a tecnologie NoSQL sono

- Migliorare la produttività dei programmatori grazie all'utilizzo di database che più si adeguano ai requisiti delle applicazioni
- Migliorare le performance in lettura e scrittura grazie alla distribuzione dei dati e al rilassamento dei vincoli di consistenza

E' essenziale verificare le aspettative sugli effettivi miglioramenti prima di adottare effettivamente le tecnologie NoSQL

L'incapsulamento di servizi favorisce la migrazione a DBMS diversi in seguito all'evoluzione delle tecnologie e dei requisiti

L'ecosistema NoSQL non è maturo come quello relazionale, ma è ancora in forte evoluzione

A meno di chiari vantaggi, si può continuare a fare affidamento sui tradizionali e robusti RDBMS