

Sistemi Informativi Evoluti e Big Data



Tecnologie per i Big Data - NoSQL

Prof. Devis Bianchini

Università degli Studi di Brescia

Dipartimento di Ingegneria dell'Informazione

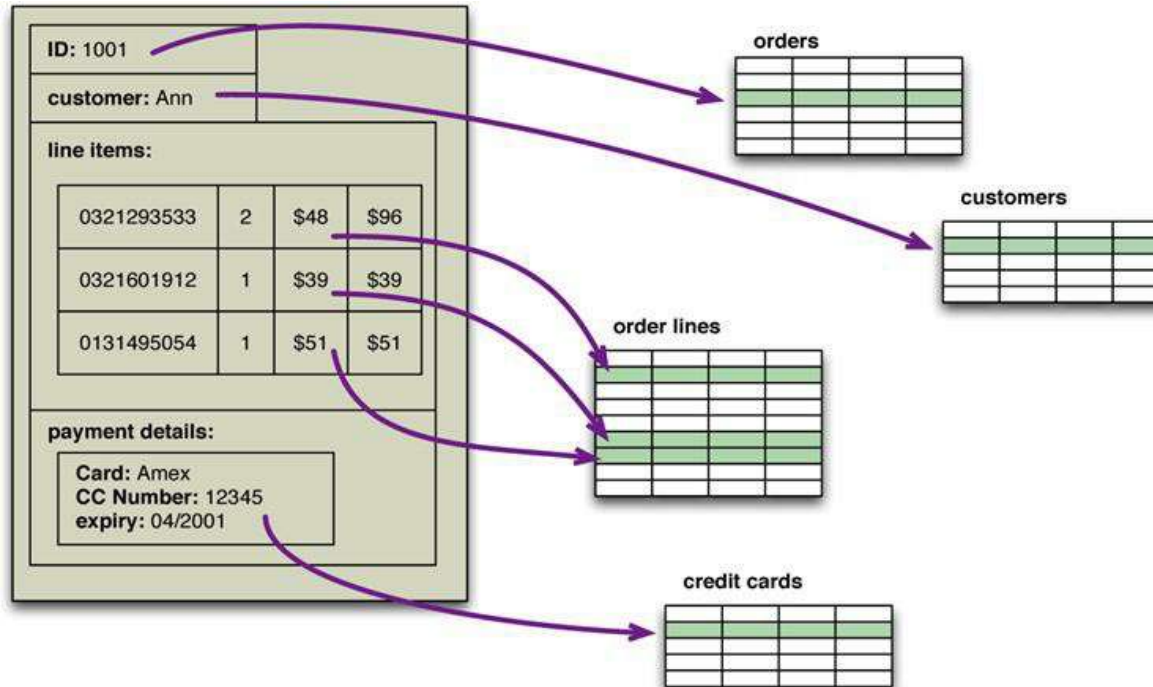


Perché NoSQL?

- Negli ultimi 40 anni i database relazionali (RDBMS) hanno costituito la tecnologia di default per lo storage e la gestione dei dati
 - Unica decisione da prendere: quale tipo di database relazionale utilizzare (talvolta nemmeno quello, in caso di fornitore dominante di DBMS relazionali all'interno dell'azienda)
- Nel passato, altri tentativi di tecnologie diverse per DBMS:
 - Database deduttivi negli anni '80
 - Database ad oggetti negli anni '90
 - Database per XML negli anni 2000
- Nessuna di queste alternative ebbe il successo sperato
 - Gli RDBMS offrono un modello dei dati standard e facile da usare (basato sul concetto di tabella o relazione) e un corrispondente linguaggio di query (SQL) efficace ed efficiente

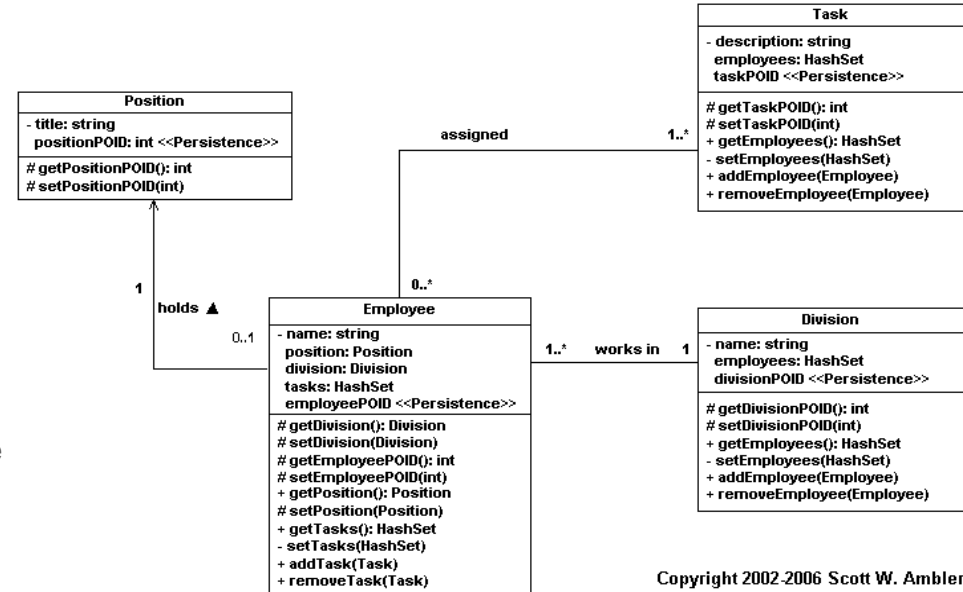
Modello relazionale e impedance mismatching

- Differenze tra il modello dei dati relazionale e strutture dati in-memory



Una proposta per risolvere il problema

- Database ad oggetti: replicare le strutture dati in-memory
- Un insuccesso, perché?
 - I database relazionali hanno avuto maggior fortuna come database centralizzati
 - Negli anni 2000 le architetture orientate ai servizi hanno portato nuove prospettive su strutture dati decentralizzate, disaccoppiando il database come sede dei dati e i data service usati verso il mondo esterno
 - Diffusione di XML e JSON



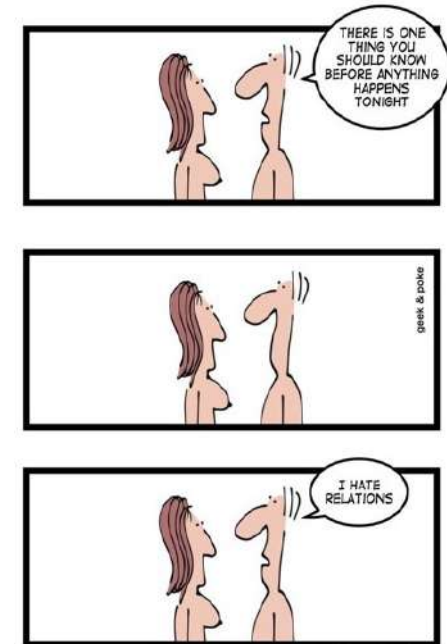
Copyright 2002-2006 Scott W. Ambler



L'avvento dei cluster

- Con l'avvento dei cluster e della scalabilità orizzontale, i database relazionali si sono dimostrati inadatti a girare su cluster di commodity hardware
- Nuove alternative al data storage
 - Google Big Tables
 - Amazon Dynamo
- NoSQL = "Not only SQL"
 - Inizialmente pensato come un modello dati basato su tabelle ASCII, no SQL
 - Da Giugno 2009 (San Francisco) una nuova prospettiva
 - Affiancamento al modello relazionale
 - Progetto open-source
 - Adatto ad essere eseguito su cluster
 - Schemaless

The Hard Life of a NoSQL Coder



Part 1: The Outing



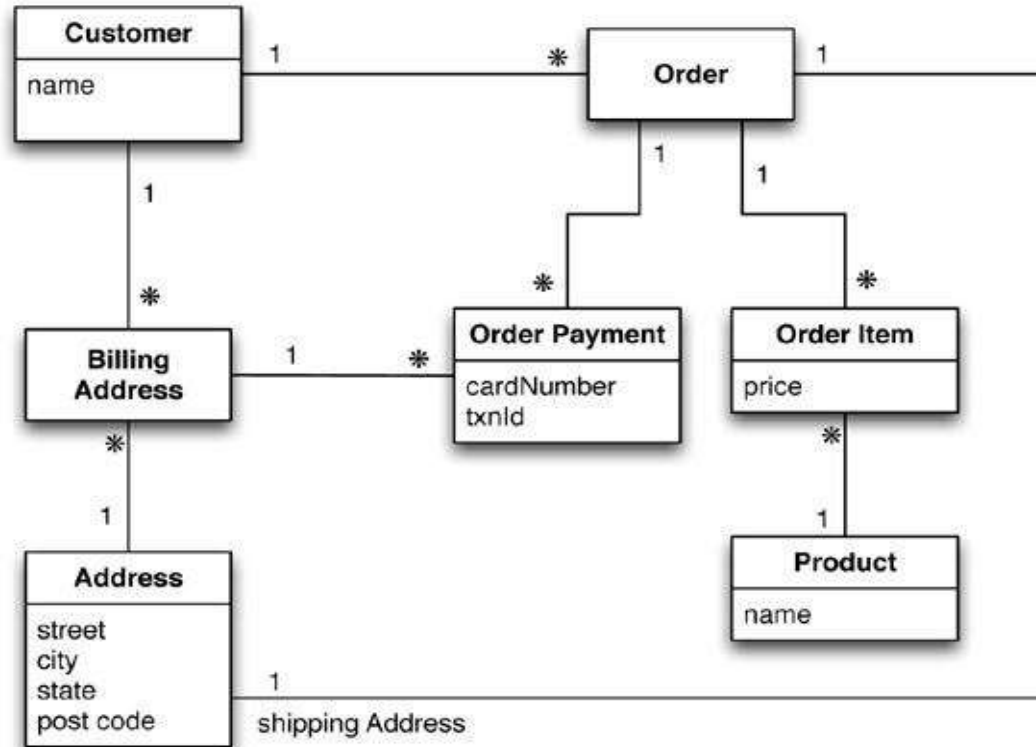
NoSQL Data Model

- *Data Model*: insieme dei costrutti per rappresentare l'informazione
 - Per esempio, nel modello relazionale: tabelle, colonne e righe
- *Storage Model*: modalità di salvataggio e gestione dei dati internamente al DBMS (tipicamente distinto e indipendente dal Data Model)
- Data Model per le soluzioni NoSQL:
 - Basato sul concetto di aggregazione
 - Key-value store
 - Document-oriented store
 - Column-oriented store
 - Basato sul concetto di grafo
 - Più vicino allo «Storage Model» rispetto al caso relazionale

Modello dei dati aggregato

- I dati come unità informativa possono avere una struttura complessa
 - Per esempio: campi di tipo diverso, array, record innestati in altri record
 - Più di un semplice insieme di record
- Aggregazione come insieme di oggetti collegati che sono trattati come unità informativa atomica, su cui agisce il sistema di storage e di gestione del dato
- Vantaggi:
 - Per gli sviluppatori delle applicazioni, che si vedono ridotto il problema dell'*impedance mismatching*
 - Per gli amministratori di DBMS, che usano tale unità informativa per la suddivisione del dato in vari nodi del cluster, al fine di garantire la scalabilità orizzontale

Modello dei dati aggregato - Esempio



Implementazione “relazionale”

Customer	
Id	Name
1	Martin

Orders		
Id	CustomerId	ShippingAddressId
99	1	77

Product	
Id	Name
27	NoSQL Distilled

BillingAddress		
Id	CustomerId	AddressId
55	1	77

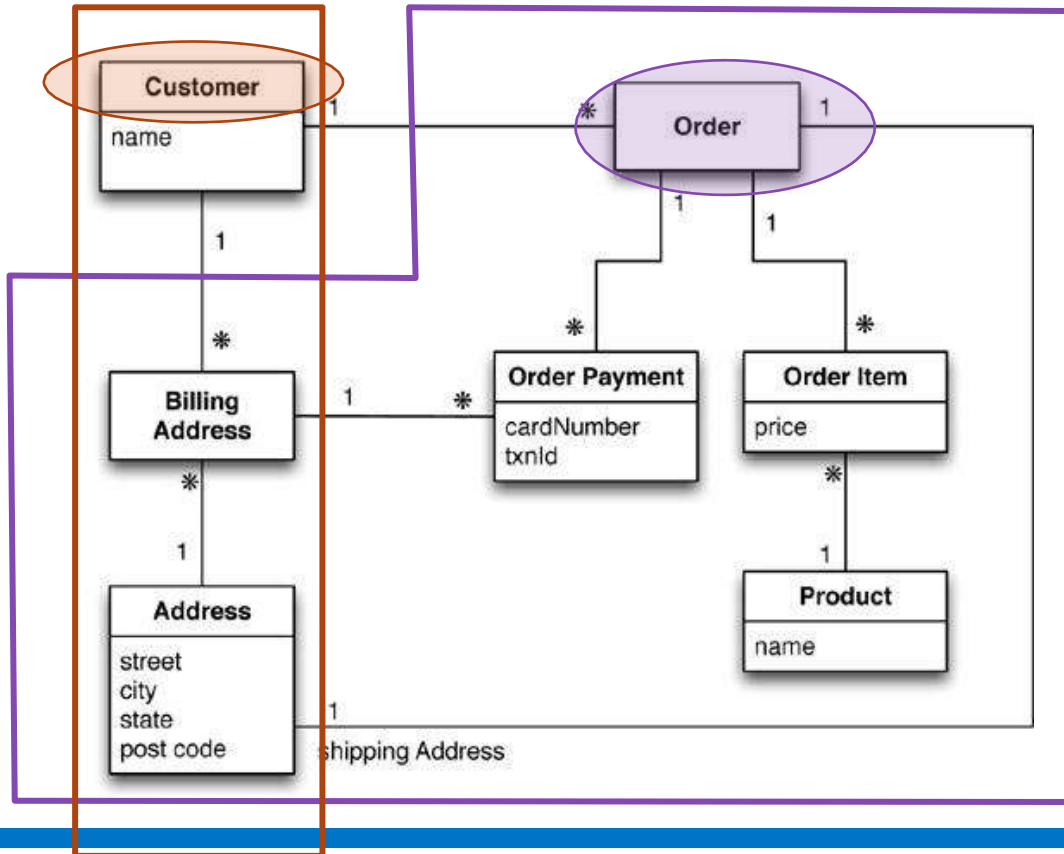
OrderItem			
Id	OrderId	ProductId	Price
100	99	27	32.45

Address	
Id	City
77	Chicago

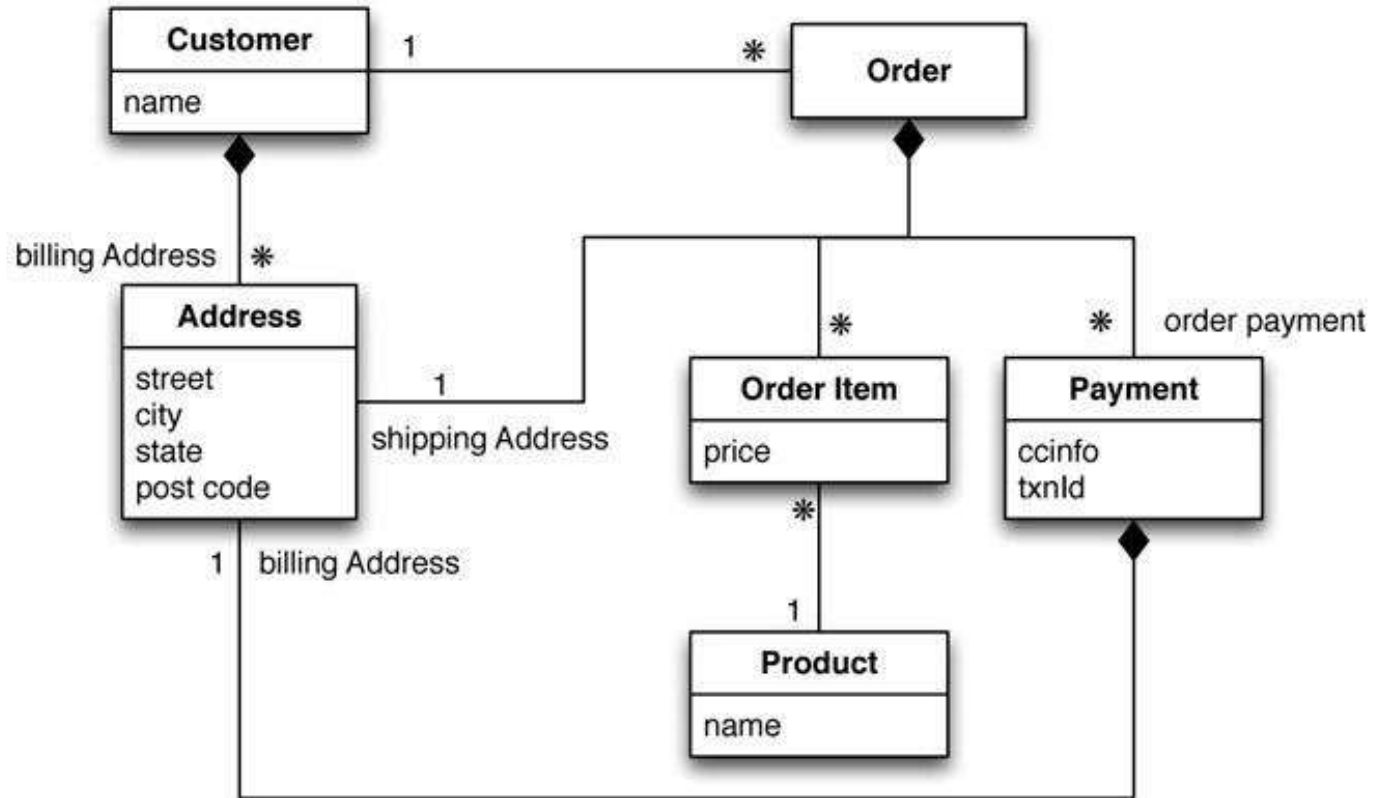
OrderPayment				
Id	OrderId	CardNumber	BillingAddressId	txnId
33	99	1000-1000	55	abelif879rft



Una possibile aggregazione



Rappresentare l'aggregazione



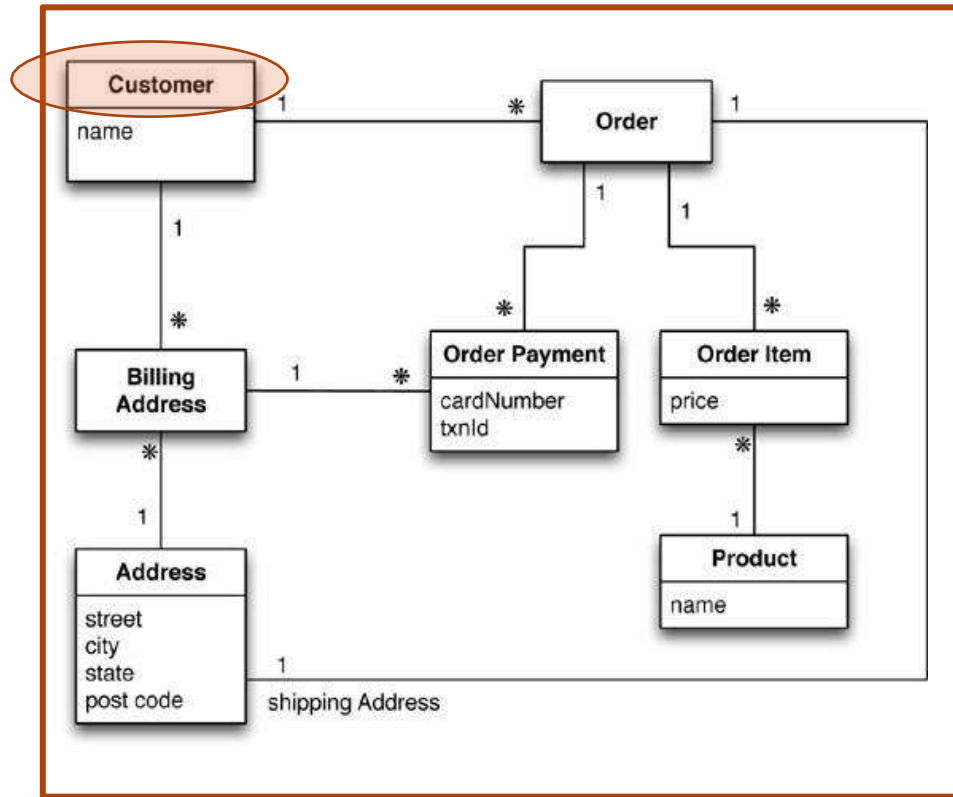
Implementazione

```
// in customers
{
  "id":1,
  "name":"Martin",
  "billingAddress":[{"city":"Chicago"}]
}

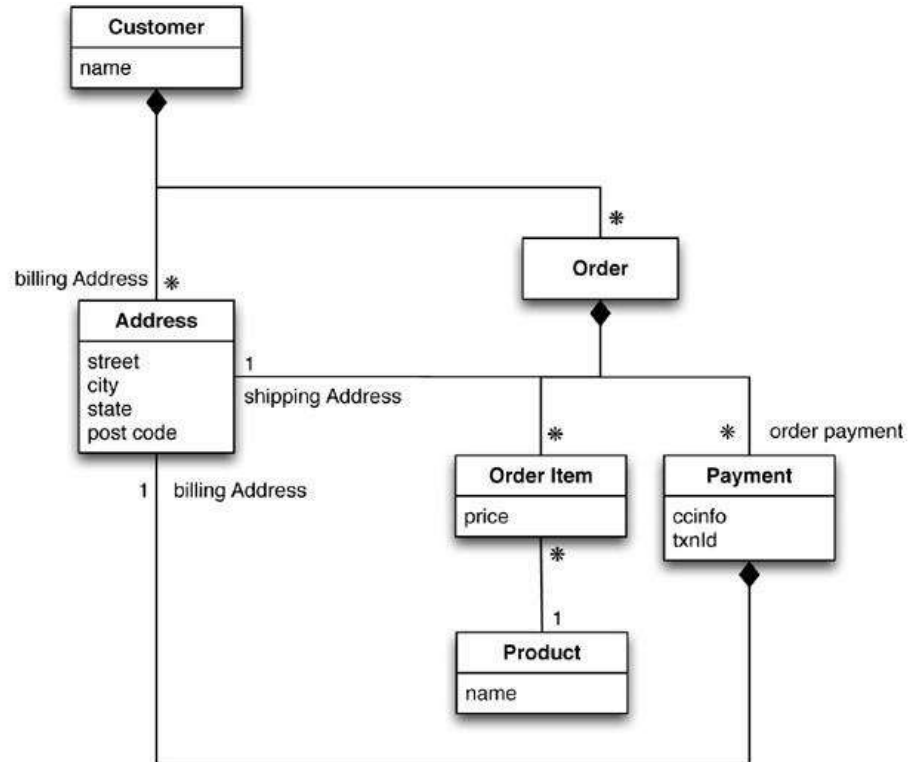
// in orders
{
  "id":99,
  "customerId":1,
  "orderItems":[
    {
      "productId":27,
      "price": 32.45,
      "productName": "NoSQL Distilled"
    }
  ],
  "shippingAddress":[{"city":"Chicago"}]
  "orderPayment":[
    {
      "ccinfo":"1000-1000-1000-1000",
      "txnId":"abelif879rft",
      "billingAddress": {"city": "Chicago"}
    }
  ],
}
```



Altra possibile aggregazione



Rappresentare l'aggregazione (II)



Implementazione (II)

```
// in customers
{
  "customer": {
    "id": 1,
    "name": "Martin",
    "billingAddress": [{"city": "Chicago"}],
    "orders": [
      {
        "id": 99,
        "customerId": 1,
        "orderItems": [
          {
            "productId": 27,
            "price": 32.45,
            "productName": "NoSQL Distilled"
          }
        ],
        "shippingAddress": [{"city": "Chicago"}]
      },
      {
        "orderPayment": [
          {
            "ccinfo": "1000-1000-1000-1000",
            "txnId": "abelif879rft",
            "billingAddress": {"city": "Chicago"}
          }
        ],
        "id": 100,
        "customerId": 1,
        "orderItems": [
          {
            "productId": 27,
            "price": 32.45,
            "productName": "NoSQL Distilled"
          }
        ],
        "shippingAddress": [{"city": "Chicago"}]
      }
    ]
  }
}
```



Strategia di design

- Non esiste una risposta universale su come aggregare i dati
- Dipende unicamente da come si intende manipolare i dati
 - Accesso ad un singolo ordine alla volta: prima soluzione
 - Accesso ai clienti insieme a tutti i loro ordini: seconda soluzione
- Forte dipendenza dal contesto
- Come detto, maggiore vicinanza allo *Storage Model*
- Vantaggi
 - Aumento dell'efficienza quando si lavora sui cluster: i dati correlati sono elaborati insieme, quindi dovrebbero risiedere sullo stesso nodo
- Svantaggi
 - Un'aggregazione potrebbe essere vantaggiosa per alcune interazioni tra i dati, ma meno per altre

NoSQL - Definizione

“Not Only SQL” – Identifica un sottoinsieme di strutture SW di memorizzazione, progettate per ottimizzare e migliorare le performance delle principali operazioni su dataset di grandi dimensioni

Perché NoSQL?

- ACID (**A**tomicity, **C**onsistency, **I**solation, **D**urability), sono le proprietà logiche che devono soddisfare le transazioni nei DB tradizionali; questo paradigma però non è dotato di una buona scalabilità
- Le Applicazioni Web hanno esigenze diverse: alta disponibilità, scalabilità ed elasticità, cioè bassa latenza, schemi flessibili, distribuzione geografica (a costi contenuti)
- I DB di nuova generazione (NoSQL) sono maggiormente adatti a soddisfare questi bisogni essendo non-relazionali, distribuiti, open source e scalabili orizzontalmente



Eventually Consistent

- **Modello di consistenza** utilizzato nei sistemi di calcolo distribuito
- Il sistema di storage garantisce che se un oggetto non subisce nuovi aggiornamenti, alla fine (quando la finestra di inconsistenza si chiude) tutti gli accessi restituiranno l'ultimo valore aggiornato
- Nasce con l'obiettivo di favorire le performance, la scalabilità e la reattività nel servire un elevato numero di richieste (rischio minimo di letture di dati non aggiornati)
- Approccio **BASE** (**B**asically **A**vailable, **S**oft state, **E**ventual consistency)
- Conosciuta con il nome di “Replicazione Ottimistica”

Approccio BASE

- Le proprietà **BASE** sono state introdotte *Eric Brewer* (Teorema CAP)
- Queste proprietà rinunciano alla consistenza per garantire una maggiore scalabilità e disponibilità delle informazioni
- **Basically Available**: Il sistema deve garantire la disponibilità delle informazioni
- **Soft state**: Il sistema può cambiare lo stato nel tempo anche se non sono effettuate scritture o letture
- **Eventual consistency**: Il sistema può diventare consistente nel tempo (anche senza scritture) grazie a dei sistemi di recupero della consistenza



NoSQL – Pro & Cons

PRO

- Schema free
- Alta Disponibilità
- Scalabilità
- API semplici
- Eventually Consistent / BASE (no ACID)

CONTRO

- Limitate funzionalità di query
- Difficoltà di spostamento dei dati da un NoSQL DB ad un altro sistema (ma il 50% sono JSON-oriented)
- Assenza di modalità standard per accedere ad un archivio dati NoSQL

Famiglie di database NoSQL

Database NoSQL basati sul concetto di aggregazione

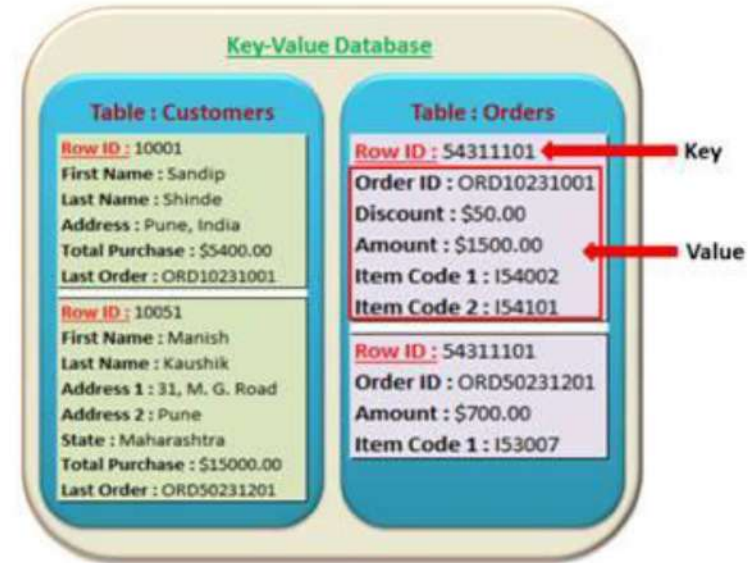
- Key-Value store
- Document-oriented
- Column-Oriented

Database NoSQL basati sul concetto di grafo

- Graph databases

Database Key-Value store (I)

- Memorizza i dati in coppie *chiave-valore* (una *chiave* è associata al resto del dato, rappresentato dal *valore*)
- Supporta l'annidamento (un valore può a sua volta contenere coppie chiave-valore)
- Richiede tanti fetch quante sono le chiavi; le chiavi corrispondono alle colonne del relazionale, ma con maggiore flessibilità
- Consente l'horizontal scaling (parallelizzazione per chiave)
- Esempi: Voldemort (LinkedIn), DynamoDB, Riak, Redis



Database Key-Value store (II)

- Per memorizzare un dato, fornire chiave e valore
 - `store.set("user-1234", "...");`
- Per leggere un dato, utilizzarne la chiave
 - `value = store.get("user-1234");`
- Le chiavi in un database di tipo Key-Value store sono memorizzate in strutture indicate con nomi diversi in soluzioni diverse (database, bucket, keyspaces, etc.); alcune soluzioni permettono anche l'enumerazione delle chiavi salvate nel database
- Per il DBMS i valori sono una sorta di BLOB, senza una struttura predefinita; quindi non è possibile validare i contenuti; i dati possono essere recuperati solo tramite le chiavi, non è possibile farlo specificando (range di) valori
- Molto efficienti per operazioni elementari come set, get, replace, del, incr, decr



Database Key-Value store – Esempi



Casi d'uso

- La tecnologia key-value store è consigliabile quando le operazioni di lettura-scrittura avvengono tramite un identificatore univoco e devono essere molto veloci:
 - Informazioni sullo stato di una sessione, profilo e preferenze degli utenti, contenuto del carrello in applicazioni di e-commerce
- L'uso di questa tecnologia è sconsigliato se:
 - Operazioni che coinvolgono più chiavi contemporaneamente, con possibili relazioni reciproche (devono essere gestite a livello applicativo e le prestazioni calano)
 - Transazioni che coinvolgono molteplici operazioni di lettura-scrittura (con requisiti di roll-back)
 - Query by data

Database Document-Oriented

- Memorizza i dati in documenti (per esempio, in formato JSON, per facilitare il salvataggio diretto di strutture dati nei diversi linguaggi di programmazione, altri formati XML e BSON)
- Record diversi ma correlati possono essere memorizzati all'interno dello stesso documento e ritornati con una sola operazione di fetch (in sostituzione del JOIN)
- Rappresenta una sorta di *sofisticazione* dei database Key-Value store
- Perfettamente adatti alla programmazione Object Oriented, eliminando il problema dell'*impedance mismatching*

Database Document-Oriented – Esempi



Casi d'uso

- La tecnologia document-oriented è consigliabile laddove la rappresentazione del dato in formato documentale (e.g., JSON o XML) è percorribile, ma c'è la possibilità che lo schema cambi in corso d'opera:
 - Event logging, piattaforme di blogging (per gestire commenti degli utenti, registrazioni, post), informazioni per applicazioni di Web Analytics, piattaforme di e-commerce dove i dati sui prodotti e sugli ordini hanno uno schema variabile nel tempo
- L'uso di questa tecnologia è sconsigliato se:
 - Sono richieste operazioni che coinvolgono più documenti contemporaneamente (devono essere gestite a livello applicativo e le prestazioni calano)
 - Le strategie di aggregazione dei contenuti nei documenti variano dinamicamente

Key-Value store vs Document-Oriented

Database Key-Value store

- Una chiave + un dato di tipo blob di grandi dimensioni senza un sistema di validazione
- È possibile salvare tutto ciò che si vuole in un aggregato
- Si può accedere ad un dato aggregato alla volta utilizzando la chiave

Database Document-Oriented

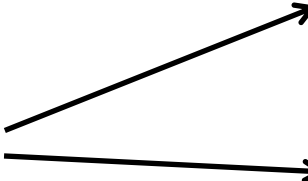
- Chiave + documento semi-strutturato
- Possono avere un sistema di validazione basato sul tipo di documento
- Maggiore flessibilità di accesso
 - Si possono sottomettere delle query in accesso basate sui campi dei documenti aggregati
 - Si può estrarre dal database parte di un aggregato anziché l'intero documento
- Indici basati sui contenuti all'interno di un documento
- Uso di collezioni per raggruppare «*documenti simili*»

Database Column-Oriented

Memorizza insieme i dati per colonna

- Scopo: minimizzare l'I/O, soprattutto nel caso di query che coinvolgono solo una parte delle colonne

RowId	Empld	Lastname	Firstname	Salary
001	10	Smith	Joe	40000
002	12	Jones	Mary	50000
003	11	Johnson	Cathy	44000
004	22	Jones	Bob	55000



```
001:10,Smith,Joe,40000;  
002:12,Jones,Mary,50000;  
003:11,Johnson,Cathy,44000;  
004:22,Jones,Bob,55000;
```

```
10:001,12:002,11:003,22:004;  
Smith:001,Jones:002,Johnson:003,Jones:004;  
Joe:001,Mary:002,Cathy:003,Bob:004;  
40000:001,50000:002,44000:003,55000:004;
```

```
...;Smith:001;Jones:002,004;Johnson:003;...
```



Database Column-Oriented – Esempi



HYPERTABLE



cassandra



Database Column-Oriented - Proprietà

- È possibile estrarre velocemente una singola colonna (o una famiglia di colonne)
- È possibile aggiungere dinamicamente colonne (risparmio in termini di tempo e di memoria allocata)
- Archiviazione differente rispetto ai RDBMS: le colonne prive di valore non vengono riportate (notevole guadagno in termini di memoria)
- Le colonne sono composte da dati uniformi (facilità di compressione e maggiore velocità di esecuzione e memorizzazione)



Casi d'uso

- La tecnologia column-oriented è consigliabile:
 - Quando serve garantire consistenza e capacità di scrittura del dato altamente scalabili (e.g., event logging)
 - In contesti analitici, per scalare linearmente al crescere del volume dei dati (e.g., per contare e categorizzare i visitatori di una pagina o portale Web)
 - Con serie di dati storici o provenienti da più fonti (sensori, dispositivi mobili), quindi spesso differenti tra loro, e con elevata velocità
- L'uso di questa tecnologia è sconsigliato se:
 - I query pattern, che coinvolgono più colonne, cambiano frequentemente nel tempo o non sono ancora del tutto chiari e definiti
 - Quando sono frequenti le operazioni di aggregazione (e.g., SUM, AVG) che coinvolgono più colonne



Aspetti chiave

- Un aggregato rappresenta una collezione di dati che viene trattata unitariamente (rappresenta i confini delle proprietà ACID di un database relazionale)
- I database Key-Value store, Document-Oriented, Column-Oriented possono essere visti come forme di database basati sul concetto di aggregazione
- Gli aggregati rendono più efficiente la gestione di grandi quantità di dati su strutture a cluster
- I database basati sul concetto di aggregazione lavorano bene quando le interazioni sui dati sono frequenti all'interno dello stesso aggregato

Relazioni

- Per realizzare relazioni tra aggregati, si utilizzano gli ID
 - L'ID di un aggregato viene inserito in un altro
 - Il database in realtà ignora che ci sia una relazione tra i dati

```
// in customers
{
  "id":1,
  "name":"Martin",
  "billingAddress":[{"city":"Chicago"}]
}
```

```
// in orders
{
  "id":99,
  "customerId":1,
  "orderItems":[
    {
      "productId":27,
      "price": 32.45,
      "productName": "NoSQL Distilled"
    }
  ],
  "shippingAddress":[{"city":"Chicago"}]
  "orderPayment":[
    {
      "ccinfo":"1000-1000-1000-1000",
      "txnId":"abelif879rft",
      "billingAddress": {"city": "Chicago"}
    }
  ],
}
```



Gestione delle relazioni

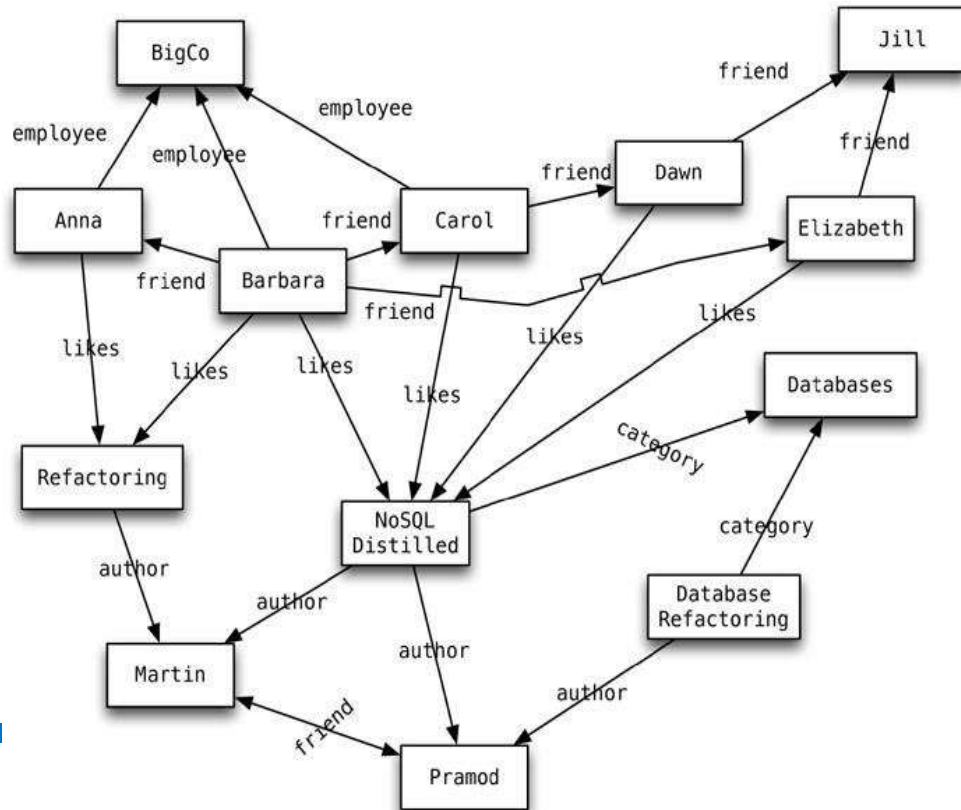
- Diversi database NoSQL non forniscono un modo per gestire le relazioni direttamente all'interno del database
 - Al massimo, i database document-oriented fanno uso di indici
- Cosa succede in caso di aggiornamento?
 - I database basati sul concetto di aggregazione trattano gli aggregati come unità su cui sono applicate le query
 - L'atomicità è garantita solo all'interno del singolo aggregato
 - L'aggiornamento di aggregati multipli è responsabilità del programmatore

Graph database (I)

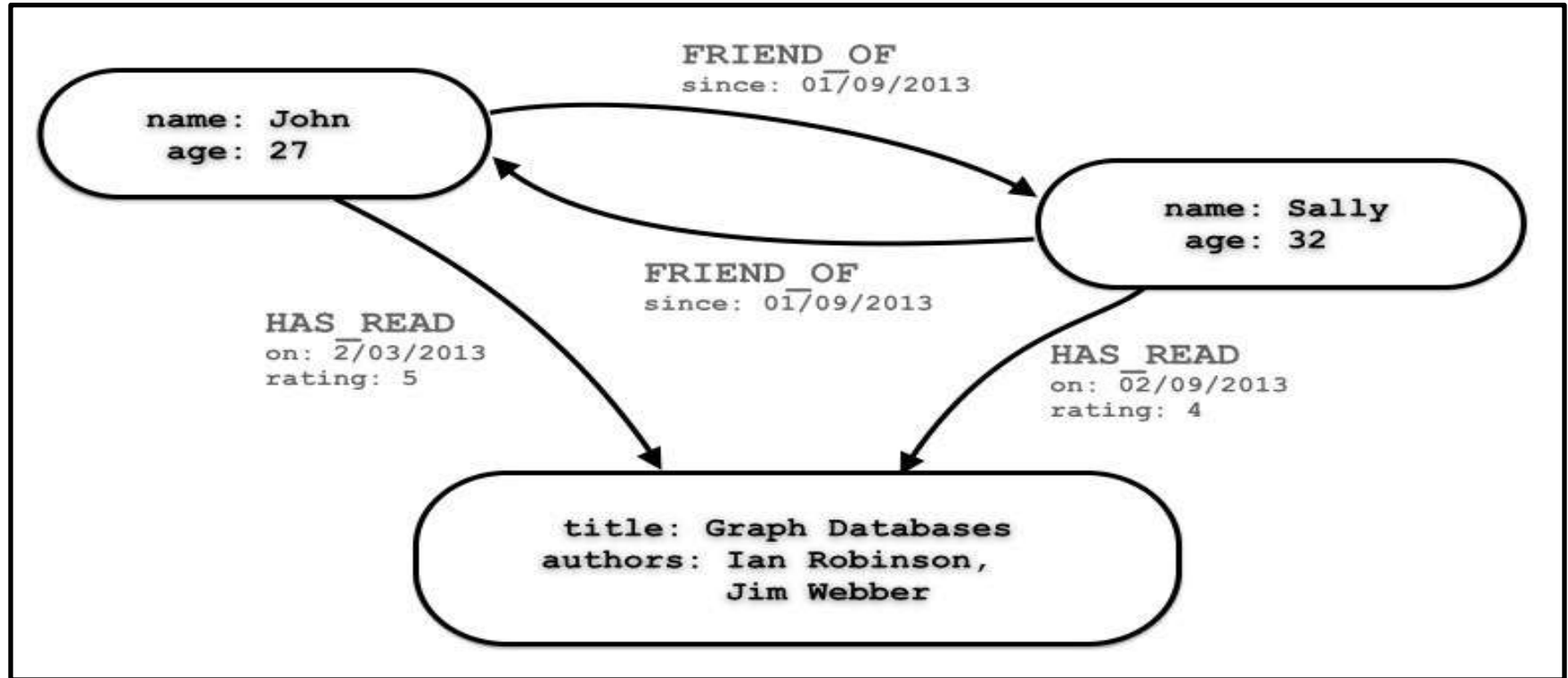
- Utilizza le strutture tipiche dei grafi (nodi e archi orientati che li connettono) e i costrutti della teoria dei grafi
- Ogni oggetto (nodo) contiene i puntatori agli oggetti (nodi) collegati
- Nodi e archi possono avere proprietà, rappresentate come coppie chiave-valore
- Quando si può usare:
 - Quando la struttura dei dati è riconducibile a quella di un grafo
 - Occupano meno spazio rispetto al volume di dati con cui sono fatti e memorizzano molte informazioni sulla relazione tra i dati
- Query come navigazione tra i nodi (e.g., individuazione di “amici degli amici”), ma più complesse rispetto ad altri database NoSQL
- Esempi: Neo4J, Apache Giraph, AllegroGraph

Graph database (II)

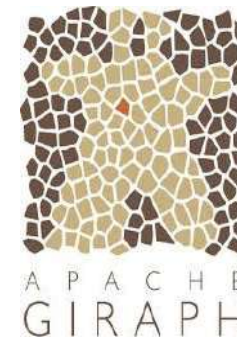
Una query possibile è la seguente: “trova i libri nella categoria Database scritti da qualcuno che piace ad uno dei miei amici”



Esempio – Graph database



Graph database – Altri esempi



Graph database vs database relazionali

- Nei database relazionali
 - Le relazioni sono implementate tramite chiavi esterne
 - Le operazioni di join navigano seguendo i vincoli di chiave esterna e sono piuttosto costose
- Graph database
 - Rendono la navigazione lungo gli archi più efficiente
 - Le performance migliorano per dati fortemente connessi
 - Il peso computazionale è spostato dalla fase di query alla fase di inserimento

Graph database vs database relazionali

RDBMS (e.g., MySQL, Postgres)	Graph Database
Tabelle	Grafi
Record/righe	Nodi
Colonne e dati	Proprietà e loro valori
Vincoli	Relazioni
JOIN	Traversal



Graph database vs aggregate-oriented db

- Modelli molto differenti
- Database basati sul concetto di aggregazione
 - Distribuiti su cluster di computer
 - Linguaggio di query molto semplice
 - Proprietà ACID non garantite
- Graph database
 - Lavorano meglio su un unico server
 - Linguaggio di query più complesso, basato su grafo



Database schemaless (I)

- I database di tipo key-value store permettono di associare qualsiasi struttura di dati ad una chiave
- I database di tipo document-oriented non impongono nessuna struttura per il documento in cui sono salvati i dati
- I database di tipo column-store permettono di salvare qualsiasi dato in qualsiasi colonna a scelta
- I database a grafo permettono di aggiungere liberamente nuovi archi e nuove proprietà ad archi e nodi



Database schemaless (I)

- Vantaggi:
 - Maggiore flessibilità (possibilità di cambiare agevolmente l'organizzazione dei dati)
 - Possibilità di lavorare con dati non uniformi dal punto di vista dello schema
- Svantaggi:
 - Un programma che accede ad un database presuppone che ci sia sempre uno schema, per poter trovare o salvare i dati che servono in un campo specificamente pensato per quello
 - Lo schema deve quindi essere gestito a livello di applicazione (per capire lo schema non si guarda il DBMS, ma il codice dell'applicazione che ne fa uso)
 - Lo schema non può essere utilizzato per decidere come salvare o recuperare i dati in maniera efficiente, né per assicurare la consistenza dei dati
 - Potenziali problemi se applicazioni multiple, sviluppate da diverse persone, accedono allo stesso DBMS



Viste materializzate

- Una vista nel modello relazionale è una tabella creata computazionalmente
- Viste materializzate: sono calcolate a priori e salvate nel database
- NoSQL database:
 - Non hanno viste
 - Possono avere query precomutate e salvate, che sono impropriamente chiamate “materialized views”
- Strategie per costruire “materialized views”
 - **Eager approach** – sono costruite e aggiornate al momento della scrittura di nuovi dati (ottimo quando le letture sono molto maggiori delle scritture)
 - **Detached approach** – procedure batch aggiornano le viste materializzate a intervalli regolari