



## **Database NoSQL – di Colonna Maurizio, Matricola: 44HHHINGINFOR**

I Database NoSQL non utilizzano il modello relazionale (a Tabelle o anche Relazioni) degli RDBMS, introdotti negli anni '70 del secolo scorso da Boyce e Codd.

La necessità di abbandonare, o meglio di ricercare altri modelli di database per gestire la persistenza e la consultazione delle informazioni (information retrieval), che non rispondessero al modello relazionale, inizia a farsi strada tra fine anni 90 e primi anni 2000, quando nascono le applicazioni Web.

Queste applicazioni sono caratterizzate dall'impredicibilità del numero di utenti (quindi introducono il problema della scalabilità) che, nel caso dei DBMS Relazionali tradizionali vengono risolti o aumentando le risorse sui server dove risiede il database oppure aumentando le risorse dei client che fanno parte della rete del/dei server. Siccome le risorse sono calcolate in base al numero di utenti (che nelle applicazioni Web è difficilmente predicibile), oppure per dimensione del database, il loro incremento ha dei forti impatti in termini di costi dell'infrastruttura, quindi parliamo di un **limite economico degli RDBMS**.

Altro limite è stata la **flessibilità**, causa la variabilità dei dati che si hanno a disposizione, i quali cambiano nel tempo. Questo è in conflitto con il paradigma dell' RDBMS che definisce prima (in modo statico) la struttura delle tabelle che conterranno i dati, quindi colonne, ed tipi di dati delle colonne. Questo ha costituito un limite per l'uso pervasivo degli RDBMS. Ultimo punto è la **disponibilità**, cioè quando si mettono in piedi architetture ad alta affidabilità occorre realizzare un'architettura con server collegati in cluster per la replica delle informazioni (ai fini dell'affidabilità), con necessità di costi aggiuntivi in termini di hardware e software. Per questo sono nate nuove architetture (in stretta relazione con il modo Hadoop ad esempio), che prevedevano l'utilizzo di sistemi commerciali e non costosi sistemi dedicati, e comunque sistemi distribuiti capaci di gestire grandi moli di dati con procedure più semplici.

Aziende come Google, Amazon o Facebook avevano **la necessità di gestire volumi di dati eccezionalmente grandi, con la necessità di avere dei tempi di risposta bassi anche con insiemi di dati enormi, pur mantenendo una elevata disponibilità del servizio**. Queste problematiche difficilmente sono risolvibili attraverso un database relazionale.

La ragione per cui un RDBMS non è in grado di gestire volumi enormi di dati deriva in parte dal modo in cui esso viene utilizzato nella maggior parte dei casi: per gestire dati organizzati su tabelle normalizzate.

**La normalizzazione dei dati** è una tecnica consolidata per la progettazione di una base dati che prevede di evitare la ripetizione dei dati su più tabelle, in modo da escludere la duplicazione di dati e rendere più semplici le operazioni di scrittura: inserire, modificare o cancellare dati diventa un'operazione molto semplice e snella, perchè i dati non sono mai ridondati su più tabelle.

Per contro, un database relazionale in cui i dati sono stati normalizzati tende a disperdere il contenuto informativo su più tabelle: ciò significa che le operazioni di recupero dei dati sparsi su più tabelle diventa più complicato, perchè è necessario recuperare le informazioni combinando i dati provenienti da molteplici tabelle (operazione di join ) ed il risultato è che i tempi di sviluppo di applicazioni che leggono dati si allungano, come pure i tempi di attesa per la lettura dei dati possono diventare eccessivamente lunghi.

Partendo da queste considerazioni, i **database NoSQL** (Not Only SQL) non prevedono il recupero di informazioni memorizzate su più tabelle, perchè l'approccio in questo caso è l'inverso: **il database deve essere denormalizzato**, cioè tutte le informazioni necessarie in fase di lettura di dati devono essere disponibili su una singola entità e questo si traduce in ridondare le informazioni su ogni entità in cui è richiesta la stessa informazione. Per esempio, se si deve visualizzare su una lista i dati relativi ad un ordine di vendita, comprese informazioni anagrafiche del cliente a cui l'ordine si riferisce, tutte le informazioni anagrafiche del cliente devono essere state salvate (duplicate) anche nell'entità dell'ordine, così da non avere bisogno di recuperare dati da altre entità.

Per contro, se le operazioni di lettura sono diventate molto più semplici e con ridotti tempi di risposta, **le operazioni di scrittura si sono complicate**: ora ogni volta che si deve aggiornare un dato ridondato in più entità, diventa necessario aggiornare il dato su tutte le entità in cui è stato duplicato.

I database No SQL hanno le seguenti caratteristiche :

sono database distribuiti

sono generalmente strumenti Open Source

sono scalabili orizzontalmente (cioè aggiungendo nodi al sistema distribuito, piuttosto che aumentando la potenza dell'hardware)

non hanno uno schema

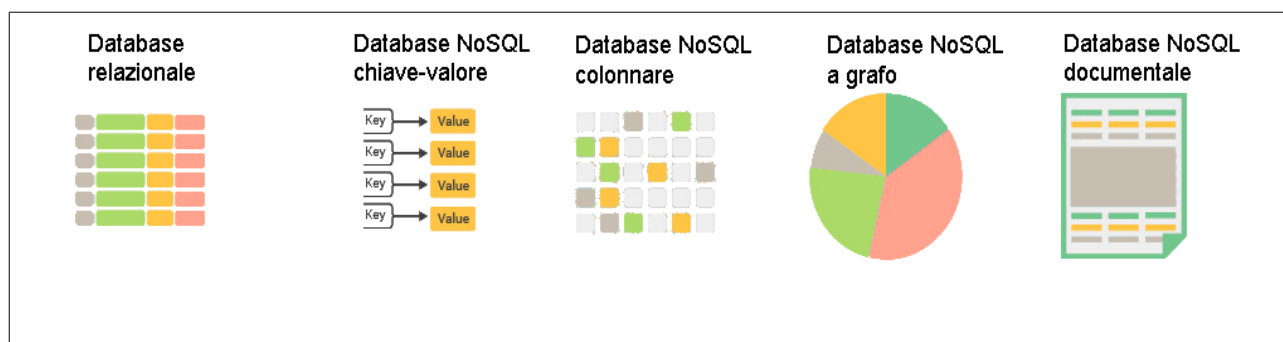
I dati sono facilmente replicabili

presentano un API semplice

non implementano le proprietà ACID delle transazioni

sono in grado di gestire moli di dati imponenti

Ci sono diverse implementazioni di **database NoSQL**, caratterizzate da approcci implementativi diversi, che si possono ricondurre a 4 categorie principali: database **chiave-valore**, **documentali**, **colonnare**, **a grafo**.



**Chiave -valore (Key-Values ) o valori associati al record.** Gestiscono coppie Chiave (identificativo del record) / valore (o valori) associate al record. Possono contenere chiavi di tipo diverso e valori di tipo diverso. Non sono quindi accoppiate fortemente, come nel caso dei database relazionali.

**Database documentali, associano a ciascuna chiave un documento.** Sono un'estensione dei database key-value, a ciascuna chiave associano un documento (un testo in JSON o XML), che possono essere strutture ricorsive.

**Database a colonna, lavorano su base colonna e non su base riga.** Sono strutture che gestiscono un numero variabile di colonne e di righe, non come gli RDBMS che hanno un numero fisso di colonne e invece righe variabili. Consentono di gestire un numero variabile di righe e di colonne, aumentando la flessibilità del sistema stesso.

**Database e grafo : nodi ed archi.** I nodi sono i dati, gli archi sono le relazioni tra i dati, sono nati in ambito social network, sono database fatti per gestire ed interrogare le relazioni ed i percorsi possibili tra i nodi. Sono importanti nei problemi di classificazione, oppure di valutazione predittiva e per relazioni causa/effetto tra i nodi.

Come concetto generale ogni DBMS deve :

Memorizzare i dati in maniera permanente (ovvero fino a quando non si scelga di cancellarli). Quindi il concetto di dati persistenti e ripristinabili a seguito di fermi macchina o guasti dell'infrastruttura.

Mantenere la consistenza dei dati (cioè a fronte di un'operazione di aggiornamento dei dati queste devono andare a buon fine oppure non essere registrate).

Assicurare l'accesso alle informazioni memorizzate, quindi la disponibilità di ogni item in esso memorizzato.

Nei database distribuiti la consistenza, la disponibilità sono invece sostituiti dal concetto di consistenza eventuale ( **eventual consistency**). Per spiegare meglio questo concetto, si riprenda una delle caratteristiche dei database NoSQL che è quella della non aderenza alla proprietà ACID (Atomitcity, Consistency Isolation Durability) delle transazioni, che può apparire come una grave mancanza, in grado di generare incoerenze nella base di dati. Questo sarebbe corretto se si avesse a che fare con un DBMS che registra i movimenti bancari su un conto corrente, quindi la proprietà ACID della transazione sarebbe fondamentale. Ma non lo è nel caso della gestione del log di un WebServer oppure di una serie di tweet. Nel mondo NoSQL l'acronimo utilizzato quando si parla di consistenza è BASE, Basically Available, Soft state, Eventual consistency : una applicazione funziona sempre (basically available), tuttavia non deve garantire la consistenza in ogni istante (Soft state), ma alla fine diventa consistente (Eventual consistency) se le attività di modifica ai dati cessano. Le inconsistenze sono transitorie e ciascun nodo del cluster alla fine otterrà le modifiche consistenti ai dati. Quindi il passo logico successivo è il

**CAP Theorem** (noto anche come Teorema di Brewer) il quale afferma che

i database distribuiti non possono avere **C**onsistency, **A**vailability and **P**artition protection tutte allo stesso tempo. **La disponibilità è la capacità di rispondere ad una qualsiasi query nel tempo. Per partizioni si intende la partizione di rete e non di disco. Se la rete che connette due o più database fallisce, i server devono continuare ad essere disponibili ed a**

ritornare gli stessi dati di prima del guasto di rete. E' un teorema importantissimo. Ci dice che, sui Dbase distribuiti si dovranno fare le scelte di compromesso, in modo da avere il giusto bilanciamento tra queste tre caratteristiche, le quali però non si possono ottenere tutte e tre assieme.

Le proprietà BASE, rinunciano in pratica alla consistenza per garantire una maggiore scalabilità e disponibilità delle informazioni.

### Tipi di Eventual Consistency

*Sono cinque tipi di consistenze (ma riconducibili a 4), comunque valide nell'ambito del concetto di Eventual consistency.*

→ Causal consistency faccio una sequenza di operazioni (lettura, scrittura etc), si mantiene il rapporto di causalità, cioè il rapporto di causa / effetto tra le operazioni perchè, per ogni modifica l'applicazione manda un segnale alle altre sessioni, che vedranno il loro dato aggiornato

→ Read-Your-Writes consistency la sessione che esegue una modifica ai dati, vedrà immediatamente tali cambiamenti, mentre le altre sessioni li potranno vedere con un breve ritardo.

→ Session consistency → all'interno di una sessione utente (login/ e poi logout), le operazioni di lettura e scrittura sono tutte coerenti tra loro.

→ Monotonic read consistency se faccio una sequenza di letture, fermo restando il contenuto del database, ottengo sempre lo stesso set di dati in risposta/risultati

◇ Monotonic write consistency se faccio una serie di operazioni in scrittura, ottengo sempre, a fronte della medesima sequenza, gli stessi risultati.

Uno dei sistemi per assicurare la consistenza è basato sui vector clock. Tale algoritmo è utilizzato per garantire la monotonic consistency, rendendo possibile l'applicazione delle modifiche ai dati, nella corretta sequenza. Tramite il vector clock è possibile tracciare quali modifiche sono state applicate ad un oggetto e chi le ha realizzate. E' come avere una sorta di array in cui sono conservati le modifiche dei vari client e l'ordine con cui sono state eseguite. Si applica ad esempio nella monotonic consistency, rendendo possibile l'applicazione delle modifiche ai dati nella corretta sequenza. Ciascun nodo del cluster gestisce un numero progressivo che indica il numero di cambiamenti (change number). Il vettore è una lista di change number che include sia quello del nodo corrente, sia quelli ricevuti da tutti gli altri nodi. Ogni volta che nel cluster accade un aggiornamento dei dati, il vettore è trasmesso a tutti i nodi, assieme all'aggiornamento. I nodi che lo ricevono, esaminano il change number ricevuto per capire se l'aggiornamento è fuori sequenza. In tal caso l'aggiornamento non è applicato, immediatamente, ma viene conservato in attesa di ricevere, e applicare, le modifiche precedenti nell'ordine corretto.

Altro esempio su come è gestita la Eventual Consistency :

**In scrittura** : *Nel momento in cui scrivo un dato in maniera replicata su vari server, l'operazione di scrittura totale non è detto che debba essere consistente. Ad esempio mentre sto scrivendo in maniera replicata il dato, un server mi va in failure e la scrittura fallisce. Allora non avrò una replica dello stesso dato su tutte le macchine in cui mi aspettavo che ci fosse. Il problema si presenterà in lettura : se un dato non lo trovo, questo significa che non ci deve essere, oppure ci sarebbe dovuto essere se il sistema non fosse andato in crash?*

**In lettura** : *Come posso decidere la consistenza in un sistema distribuito ? Si introduce il concetto di quorum, che è il numero di server che devono rispondere ad una operazione di read o write affinché possa essere considerata completa. E' un concetto nuovo,*

*introdotto dai sistemi distribuiti. Immaginiamo un database distribuito su 5 server. Se faccio una operazione di lettura e quasi contemporaneamente ne parte una altra di scrittura sullo stesso database, potrei definire un quorum per considerare completata la richiesta di scrittura in 3 (cioè completata in 3 database dei 5 totali del cluster). Allo stesso modo se, con una richiesta di lettura, ottenessi lo stesso dato da 3 server ho una conferma del quorum sulla consistenza del dato. Quindi il quorum può essere definito in modo variabile, in particolare nel caso dei 5 server, sarà un numero che varia da 1 a ... 5.*

La scalabilità orizzontale dei database NoSQL consente di poter fare a meno delle prestazioni e delle funzionalità di alta disponibilità fornite da hardware ad alto costo, utilizzando invece hardware commerciale a basso costo. I nodi di un cluster su cui è installato un database NoSQL possono essere aggiunti o rimossi senza particolari problematiche di gestione, realizzando così una scalabilità orizzontale a costi contenuti degli ambienti Multi-Nodo. Con lo sharding dei dati, mediante opportuni algoritmi i dati vengono suddivisi e distribuiti su più nodi, recuperandoli quando necessario (es. algoritmo di gossip). I dati distribuiti sui vari nodi, spesso sono anche copiati un certo numero di volte per garantire la disponibilità delle informazioni.

I NoSQL DB sono oggetto di continui studi al fine di migliorare

**Performance :** *Vengono studiati e implementati dei nuovi algoritmi al fine di aumentare le prestazioni complessive dei sistemi NoSQL.*

**Scalabilità orizzontale :** *è fondamentale riuscire ad aumentare/diminuire la dimensione di un cluster aggiungendo/rimuovendo nodi in modo "invisibile"; cioè l'intero sistema non deve fermarsi o accorgersi di cosa sta accadendo.*

**Performance sulla singola macchina :** *è importante riuscire ad aumentare anche le prestazioni di ciascuna macchina, poiché spesso sono responsabili di ricercare informazioni nel cluster a nome di applicativi esterni.*

*Note sulle garanzie di Consistenza e "Durability"/"Persistenza" dei dati ne DB NoSQL*

Una delle caratteristiche dei DB NoSQL è, contrariamente a quanto avviene per i DB tradizionali, la possibilità di scegliere il livello di Persistenza e di consistenza dei dati a qualsiasi livello del sistema.

Riprendendo il discorso del Replication Factor, se questo è ad esempio pari a 3, ciò significa che una volta che il dato è stato scritto nel nodo master, sarà poi scritto nei nodi di replica.

Chiaramente queste operazioni aggiuntive di scrittura hanno un overhead, specialmente se i nodi di replica sono separati fisicamente dal Master Node, e quindi si dovranno considerare i tempi di latenza della rete (network latency).

In alcuni casi noi potremmo non volere che le nostre transizioni attendano il completamento in scrittura delle repliche, quindi possiamo modulare le nostre politiche di persistenza ed adattarle ad altri parametri.

Se scegliamo ad esempio per la persistenza il valore di 1, quando il dato è scritto nel Master Node l'operazione ritornerà il controllo al programma chiamante e la replica dei dati si effettuerà in background.

Nei database NoSQL Oracle, i modelli di persistenza hanno 3 livelli di riconoscimento :

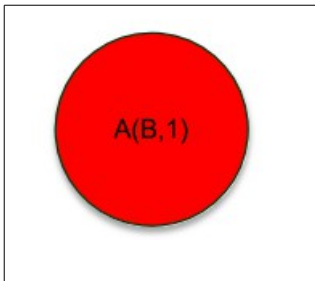
- *Master Node only*
- *All Nodes (presenti nell'insieme di replica) – che applicano la replica sincrona*
- *A majority of nodes (Una maggioranza di nodi nell'insieme di replica )*

In più, le policy nei Db NoSQL Oracle permettono il controllo a livello di persistenza per le operazioni in scrittura nei nodi Master ed in quelli di Replica. Questo può essere realizzato scegliendo se il dato sarà scritto in

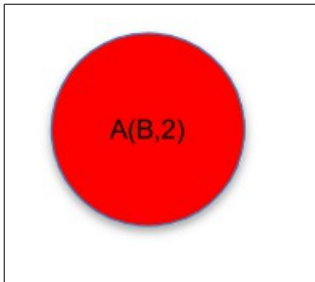
- a) un buffer della memoria locale
- b) nella cache buffer del Sistema Operativo
- c) attende che sia scritto comunque su disco

Queste caratteristiche introducono una grande flessibilità nelle performance della persistenza. La possibilità di controllare questo a livello delle transazioni consente, per alcune operazioni, di essere eseguite in regime di “sicurezza dall'errore” mentre altre possono sacrificare la persistenza in favore delle prestazioni.

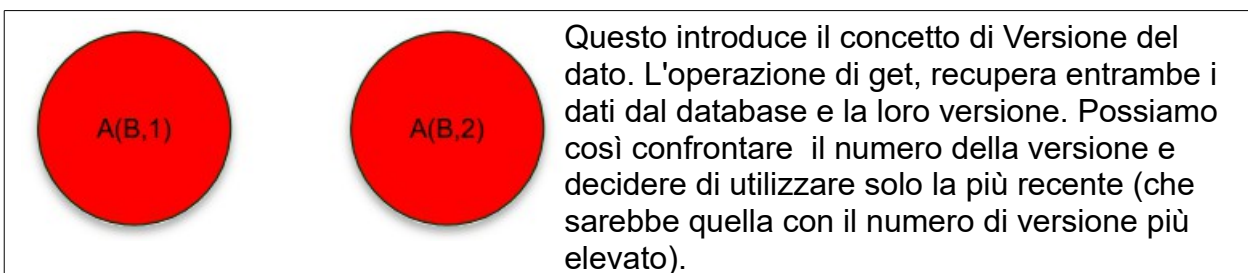
Il concetto di persistenza si lega poi a quello di Consistenza (e versione del dato), che viene gestito diversamente da quanto implementato negli RDBMS tradizionali. Quando introduciamo un dato nel KV Store è implicito attribuirgli una versione nel sistema, vediamo il motivo con un semplice esempio. Noi inseriamo una coppia chiave-valore (KV pair) con chiave/Key A e valore/Value B ed è assegnata la versione 1, che possiamo rappresentare così :



Ora, immaginiamo un processo che nel suo sviluppo, arriverà ad aggiornare quel dato coppia chiave/valore che era stato inserito, e che andrà a variarne il valore. Quello che è veramente importante non sarà il nuovo valore ma il fatto che si dovrà aggiornare la versione del dato, per registrarne il cambiamento :



Fino a qui tutto bene, ma come si lega questo con l'utilizzo dei dati del database ? Ad esempio se noi abbiamo delle regole di persistenza meno stringenti, che consentono l'aggiornamento in background delle repliche (quindi repliche asincrone del dato), questo introduce un problema di consistenza del dato stesso, perchè ad esempio dal Master Node leggeremo il dato in versione 2, ma se lo leggiamo da un replica Node, potremo leggere con molta probabilità la versione 1 del dato stesso, quindi come gestiamo questo disallineamento dei dati ?





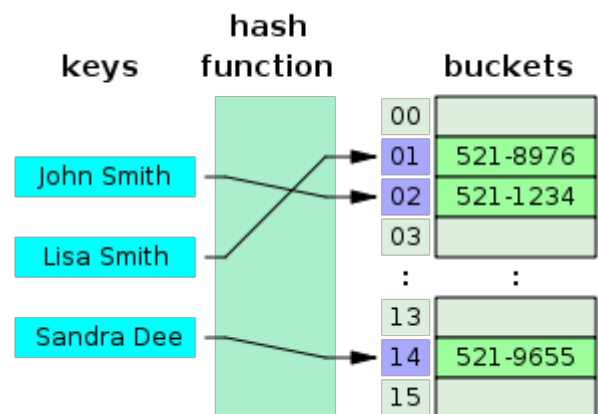
## Tipologie di Database NoSQL

La prima categoria presa in esame è quella dei **Key/value database (chiave/valore)**. Sono basati sul concetto di *associative array* (o dictionary o map). I **vettori associativi**, sono un concetto molto più flessibile ed ampio, dei vettori classici e sono una struttura in grado di contenere un insieme di coppie chiave/valore. In generale la chiave è un valore univoco con cui è possibile ricercare un valore nel database. Gli associative array sono di solito implementati con le hash table. Le chiavi devono essere uniche e sono stringhe Java. Una chiave può essere qualsiasi cosa a noi necessaria per rappresentare e gestire una determinata informazione. E' una soluzione che permette di ottenere una buona velocità ed una eccellente scalabilità orizzontale (parallelizzazione dei nodi).

Diverse combinazioni di coppie chiave-valore danno origine a diversi database



E' di fondamentale importanza la progettazione della chiave, poiché lo storage si basa su indirizzamento diretto. La chiave viene trasformata, attraverso l'utilizzo di una funzione di Hashing, in una chiave HASH. Si ottiene una MKC (major key component) ed opzionalmente in una o più mkc (minor key component). Data quindi una java string che rappresenta una key, viene convertita dalla hash function in una MKC ed in una o più mkc. Perché viene fatto questo ? Perché le chiavi sono sparse tra le partizioni (dette Shards) utilizzando come hash la MKC. Le Shards sono una partizione orizzontale dei dati nel database. Ogni Shard è gestita da un determinato Database Server ed è identificata da una MKC. Questa è una modalità di partizionamento cosiddetta orizzontale, che è diversa da quella classica dei database relazionali (che è una partizione verticale), in cui i dati sono partizionati su server diversi ma per colonna.



La hash table è una struttura dati costituita da un array di valori e da una funzione di hashing che consente di mappare ogni chiave a una posizione dell'array. Le funzioni di hashing trasformano un dato a lunghezza variabile in un altro dato a lunghezza fissa, per esempio una stringa in un numero intero. Attraverso la funzione di hashing è quindi possibile trovare immediatamente un valore in base ad una chiave. La funzione di hashing deve essere deterministica, ovvero a parità di input, deve ritornare sempre lo stesso output, altrimenti non sarebbe possibile determinare con certezza la posizione di una chiave nell'array.

Un KVStore consiste di una molteplicità di componenti, ma cercando di

descriverlo ad un livello di astrazione più alto, possiamo dire che è composto da molti nodi di memorizzazione (storage nodes). Uno storage node è un server (o una macchina virtuale), con dei dischi locali. In uno storage node ci sono i replication Nodes, il cui numero non è determinato a priori, ma può variare in funzione delle necessità di memorizzazione. Ogni replication Node ospita una o più partizioni (almeno sempre più di una partizione).

Il Partizionamento (Partitioning) ovvero : come decide il sistema di distribuire i dati (i valori) utilizzando il sistema delle chiavi ?

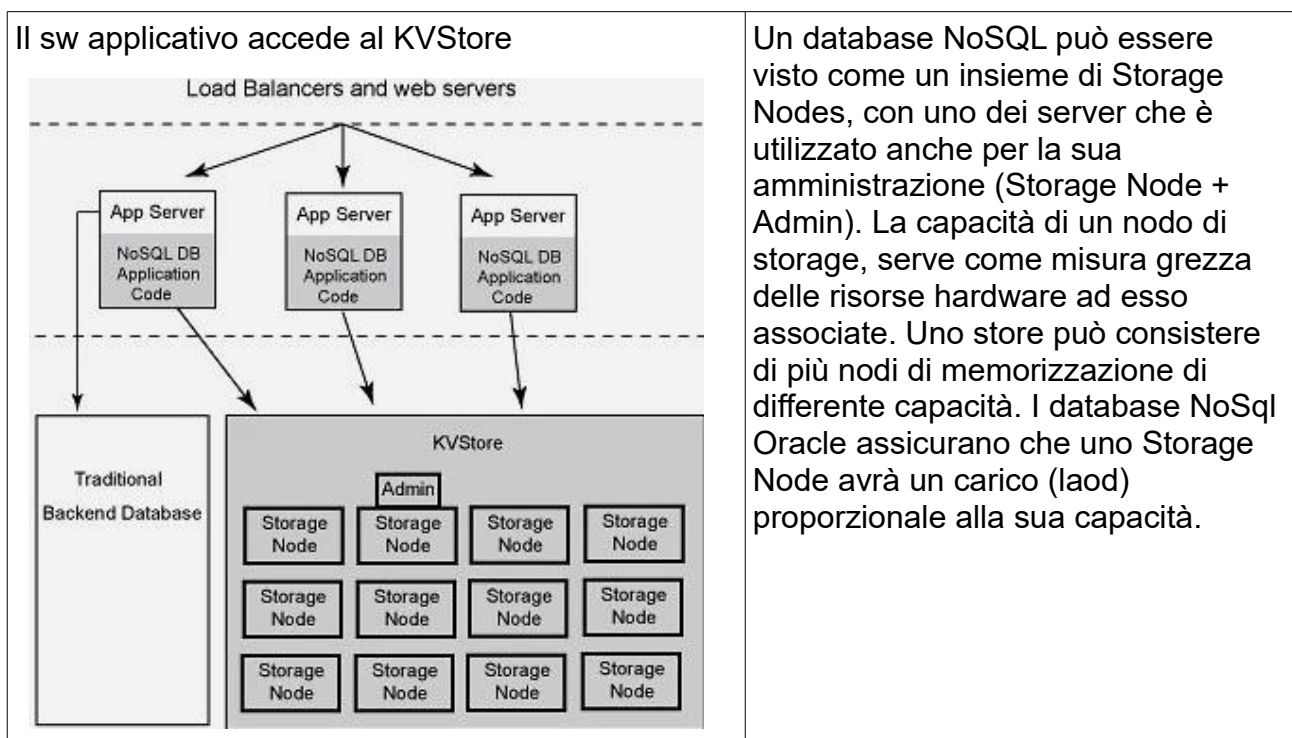
Quando un valore è memorizzato viene utilizzato un algoritmo di hashing il quale, in funzione della chiave in input, restituisce un partition ID che è la MKC, la quale, in combinazione con la mkc individua il dato all'interno della partizione.

La configurazione architetturale prevede queste caratteristiche :

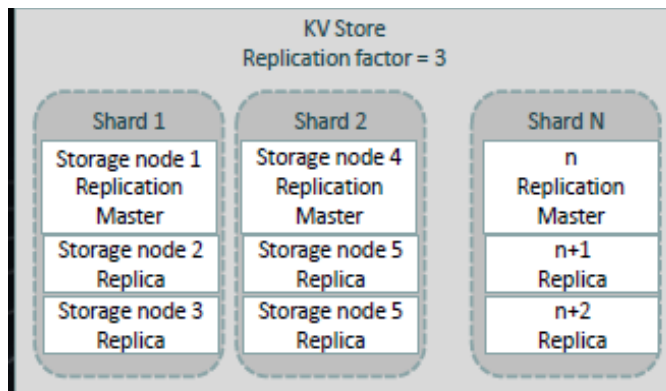
I database Kvalue (KVStore) sono un insieme di

- Storage Nodes che ospitano un insieme di
- Replication Nodes
- I dati sono sparsi attraverso i Replication Nodes
- Un Replication Node può essere pensato come un singolo DB che contiene le coppie KV (Chiave / Valore)
- Ogni Storage Node è ospita uno o più Replication Nodes, ma questo è determinato dalla sua capacità
- Un Replication Node a sua volta contiene almeno una ma in generale più partizioni e una singola partizione conterrà più chiavi.

Uno schema di questa architettura è il seguente :







Gli Storage Node contengono una serie di replication node, Se nella shard il Replication Factor è =3, allora avremo 1 Replication Master (che si occuperà di scrivere le repliche) e 2 Storage di Replica.

Nel KVStore abbiamo i macro raggruppamenti detti Shard, che uniscono più nodi. Il numero di nodi che fanno riferimento ad una shard è chiamato Fattore di Replica (Replication Factor) dei dati stessi. Maggiore è il Replication factor e più veloce è la lettura dei dati, ma la scrittura ha delle performance più basse, perchè un'operazione di scrittura necessita di più tempo di una in lettura, e queste operazioni aumentano col il fattore di replica.

## Key-Values Data Manipulation

Gli associative array consentono le seguenti operazioni :

*Add* : si aggiunge un elemento all'array

*Remove* : operazione con cui si toglie un elemento dall'array

*Modify* : si cambia il valore associato ad una certa chiave

*Find* : si ricerca un valore in un array tramite chiave

tali operazioni sono le stesse che i database della categoria key/value store consentono sui dati. Si tratta di operazioni molto semplici, che garantiscono però tempi di esecuzione costanti.

Se analizziamo ad esempio la differenza tra un DB tradizionale (SQL) ed un DbNOSQL, sulla selezione di un certo tipo di informazione :

```
SELECT address, city, country
FROM
```

```
Customer
```

```
WHERE
```

```
city = 'Bristol' --> Voglio selezionare le colonne address, city e country dalla tabella
customer in cui la città è Bristol (quindi seleziono dalla tabella Clienti quelli che sono della
città di Bristol e ne visualizzo Indirizzo, Città e Paese). Questa è una classica SELECT di
un RDBMS relazionale.
```

Per ottenere lo stesso risultato da un DbNOSQL di tipo KV dovrò scrivere del codice che mi effettua una scansione iterativa con dei parametri di ingresso, che mi restituisce una lista di occorrenze:

```
appData[cust:2433:address] = '258 Camberley rd, Bristol, UK'
```

```
define findCustomerWithCity(p_startID, p_endID, p_City):
```

```
begin
```

```
    returnList = ();
```

```
    for id in p_startID to p_endID:
```

```

        address = appData['cust:' + id + ':address'];
    if inString(p_City, Address):
        addToList(Address,returnList );
    return(returnList);
end;

```

I valori sono degli array di byte e non richiedo una tipizzazione forte, ad esempio possiamo avere una stringa non tipizzata come la seguente : ('258 Kew rd','Richmond','Surrey','UK')

oppure una stringa tipizzata : {'Street':'258 Kew rd','City':'Richmond','County':'Surrey','Country':'UK'}, con associati gli attributi di ogni valore, quindi la Street, la City e così via..è simile al concetto di colonna degli RDBMS tradizionali

Oppure possiamo utilizzare gli schemi AVRO, per definire dei valori. Si definiscono degli schemi, all'interno del valore e si può tipizzare il contenuto del valore stesso. I tipi AVRO fanno parte dell'ecosistema APACHE, eccone un esempio creato usando un formato JSON (che sarà memorizzato in un file specifico, ad esempio *PersonSchema.avsc*) :

```

{
    "type": "record",
    "namespace": "FVAvro",
    "name": "PersonInformation",
    "fields": [
        { "name": "first", "type": "string" },
        { "name": "last", "type": "string" }
    ]
}

```

Si potrà memorizzarlo nello store con il comando **ddl add-schema**

→ kv-> ddl add-schema -file PersonSchema.avsc

- Per rendere lo schema disponibile nel codiceo basterà leggerlo così :
- Parser parser = new Schema.Parser();
- parser.parse(new File("PersonSchema.avsc"));
- Come passo successivo, abbiamo bisogno di rendere disponibile lo schema alla nostra applicazione :
- personSchema = parser.getTypes().get("FVavro. PersonInformation" )
- Quindi abbiamo bisogno di utilizzare nella nostra applicazione, i campi presenti nello schema e per farlo dobbiamo creare un collegamento (binding), traducendo dal valore allo schema Avro
- Quindi si potranno controllare/vedere i valori dei dati utilizzando i campi dei record Avro.

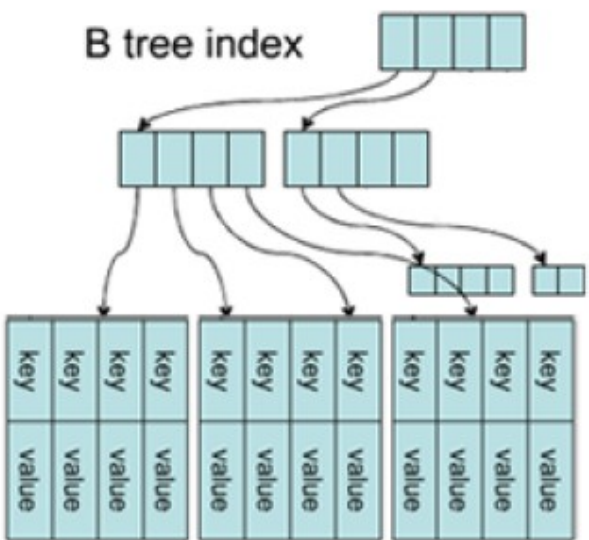
Tra i KVStore NoSQL abbiamo :

Berkley DB : è un KV DB ad alte prestazioni , attualmente gestito ed aggiornato da Oracle, Il nome Berkley DB indica in realtà tre diverse implementazioni

- > Berkley DB : key/value store scritto in linguaggio C
- > Berkley DB Java Edition (JE) : key/value store scritto in Java, facilmente integrabile in applicazioni java.
- > Berkley DB XML : scritto in C++, utilizza il key/value store oer l'indicizzazione e l'ottimizzazione di uno storage XML

Berkley DB consente il salvataggio delle coppie chiave/valore in quattro diverse strutture dati :

- *B-tree* : è una struttura dati (albero bilanciato), utilizzata negli RDBMS per memorizzare gli indici delle tabelle. Queste strutture mantengono gli elementi ordinati e consentono un l'accesso alle informazioni (le performance non sono però ottimali)

	<div style="border: 1px solid red; padding: 5px; text-align: center;">550e8400-e29b-41d4-a716-446655440000</div>
<p>Alcuni Key-Value DB permettono la definizione di indici secondari (es. B+tree) per le chiavi, ma si hanno perdite di performance e problemi nella gestione del clustering</p>	<p>Meglio utilizzare chiavi naturali o meccanismi distribuiti per la loro generazione come UUID (Universally Unique Identifier) - 32 caratteri esadecimali (8 - 4 - 4 - 4 - 12)</p>

- *Queue*. La coda è una serie di record sequenziali a lunghezza fissa. Le chiavi sono costituite da record number di tipo intero e le scritture sono molto veloci perchè i record sono aggiunti in sequenza sulla coda.

- *Recno* : è una struttura simile alle code, ma ne differisce perchè i record sono a lunghezza variabile.

In Berkley DB è possibile accedere ai dati attraverso due differenti modalità. La prima consiste nell' utilizzo dell' API di basso livello, che fornisce metodi come put e get (per salvare e recuperare i dati) e che offre il massimo controllo sui dati. Il secondo metodo prevede l'utilizzo del Direct Persistent Layer (DPL), che gestisce la persistenza a livello di classe sul database Java, attraverso l'annotazione della classe stessa e delle sue proprietà. Le caratteristiche principali di Berkley DB sono :

- Possibilità di utilizzare dati complessi sia come chiave sia come valore
- Il supporto alle transazioni, che può essere attivato o disattivato
- Supporto ad indici primari e secondari
- La possibilità di replicare il database su più nodi
- La possibilità di gestire grandi dataset (il limite è 256 TB)

La replica consente accessi più veloci ai dati, ma non permette la scalabilità in base al volume dei dati, poichè la replica consiste nella copia di tutto il database in un altro server e questo, sembra essere una caratteristica limitante.

Dal berkley DB sono stati costruiti numerosi altri database NoSQL basati sul concetto di key/value store tra cui : Project Voldemort (sviluppato da LinkedIn), DynamoDB di Amazon, Tokio, Riak.

Questo tipo di database NoSQL è indicato per la gestione liste con un numero elevato di elementi, di informazioni delle sessioni delle web app, degli "shopping cart" per gli acquisti online (con volumi di dati variabili nei diversi periodi dell'anno).

## **Column-oriented database**

L'organizzazione dei dati per colonna, invece che per riga, non è un modello presente solo nel movimento NoSQL, ma è ampiamente utilizzato nella Business Intelligence e nei cosiddetti analytics.

I punti di forza di un data store colonnare, consistono nell'evitare di sprecare spazio quando un determinato valore non esiste per una certa colonna e di consentire un elevato grado di compressione dei dati. Nei database organizzati per riga invece, anche i valori nulli occupano spazio.

Il modello di archiviazione utilizzato dai column store database, prevede l'utilizzo di coppie chiave/valore (o colonna/valore), come accad nei key/value store. Tuttavia una singola unità di dato contiene più coppie chiave/valore ed è identificata da una chiave univoca detta chiave primaria. Le unità di dati sono salvate in ordine di chiave primaria. Il modello è molto flessibile, perchè ciascuna riga può contenere una o più colonne (al limite tutte).

Il database colonnare di riferimento è BigTable di Google che, nel paper del 2006 lo definì come "una mappa multidimensionale persistente ordinata scarsamente distribuita", nel quale i valori sono salvati come array di byte e sono identificati da un indirizzo composto dalla chiave di riga (row key) e dalla chiave di colonna (column key) e dal timestamp. Le chiavi di riga sono stringhe univoche, mentre il timestamp è un intero a 64 bit che serve per differenziare le diverse versioni dell stessa cella (insieme riga/colonna). Il partizionamento è effettuato sulla base della row key e l'unità di partizionamento è chiamata tablet. Per ciascuna tabella le colonne, il cui numero non è limitato, sono organizzate in column family, che devono essere predefinite prima dell'inserimento delle colonne stesse. Le column family costituiscono l'unità di base per la gestione degli accessi e la compressione dei dati in BigTable.

Il file system di gestione di BigTable è il Google File system (GFS), che è il file system distribuito di Google per la persistenza dei dati. Inoltre, per assicurare la consistenza del sistema, BigTable utilizza Chubby, un servizio di locking distribuito con la caratteristica dell'alta disponibilità, basato sull'algoritmo Paxos.

Paxos è un algoritmo di consenso, utilizzato ad esempio per operazioni di replica di database. Per consenso si intende il processo con cui un gruppo di partecipanti si accorda sul risultato di un operazione.

Vediamo uno schema su Column-Oriented Database, anche in confronto con database tabellari tradizionali :

In un DBMS relazionale una ipotetica tabella anagrafica avrebbe la seguente struttura:

ID	COGNOME	NOME	DATA DI NASCITA
1	Rossi	Mario	01/01/1960
2	Verdi	Giuseppe	01/01/1961
3	Bianchi	Antonio	01/01/1962

Le informazioni sarebbero memorizzate riga per riga nel seguente modo:

- 1, Rossi, Mario, 01/01/1960;
- 2, Verdi, Giuseppe, 01/01/1961;
- 3, Bianchi, Antonio, 01/01/1962;

In un DB Column Oriented la stessa tabella anagrafica avrebbe la seguente struttura:


ID	COGNOME	NOME	DATA DI NASCITA
1	Rossi	Mario	01/01/1960
2	Verdi	Giuseppe	01/01/1961
3	Bianchi	Antonio	01/01/1962

le stesse informazioni vengono divise in colonne:

- 1, 2, 3;
- Rossi, Verdi, Bianchi;
- Mario, Giuseppe, Antonio;
- 01/01/1960, 01/01/1961, 01/01/1962;

La tabella memorizza una colonna alla volta e alla fine procede con la memorizzazione della colonna successiva.

Oltre alla colonna è memorizzata anche una chiave surrogata necessaria per **ricostruire il record**.

<p>I database chiave-valore possono evolvere in column-family database (composti da gruppi di colonne), in cui i dati sono memorizzati per colonne invece che per righe.</p>	
	<p>I dati sono organizzati ancora come una hash table, ma si usano due o più livelli di indicizzazione.</p>

La column-family è un'informazione che deve essere presente nello schema. Raggruppa un insieme di colonne appartenenti alla stessa tipologia di informazioni, Es. "anagrafica" (Nome, Cognome, Età...)

Le colonne che costituiscono una column-family non devono essere definite a priori.

Ogni record potrà avere informazioni differenti all'interno di una column-family.

Le colonne vuote, per le quali non si ha alcun valore, non saranno presenti all'interno di una column-family.

Esempio. Consideriamo un caso d'uso con due Unità Informative (UI), nello specifico due persone, per le quali abbiamo informazioni memorizzate all'interno di due Column Family (CF) :

**anagrafica                      indirizzo**

**UI : 1**

**CF : anagrafica**

*nome → Arturo | cognome → Collodi | genere → Maschile | eta → 39*

**CF : indirizzo**

*via → Pinocchio 1 | cap → 20121 | città → Milano | nazione → Italia*

**UI : 2**

**CF : anagrafica**

*nome → Sara | cognome → Piccolo | genere → Femminile | eta*

**CF : indirizzo**

*via → Alfieri 2 | cap → 20134 | città → Milano | nazione*

Archiviazione differente rispetto ai RDBMS. Le colonne prive di valore non vengono riportate.

Notevole guadagno in termini di memoria, significativo soprattutto su grandi numeri.

**CF : anagrafica**

UI : 1 *nome → Massimo | cognome → Carro | genere → Maschile | eta → 39*

UI : 2 *nome → Sara | cognome → Piccolo | genere → Femminile | peso ->55*

- Aggiunta della chiave hobby e eliminazione del genere nell'UI 1

**CF : anagrafica**

UI : 1 *nome → Massimo | cognome → Carro | hobby → lettura | eta → 39*

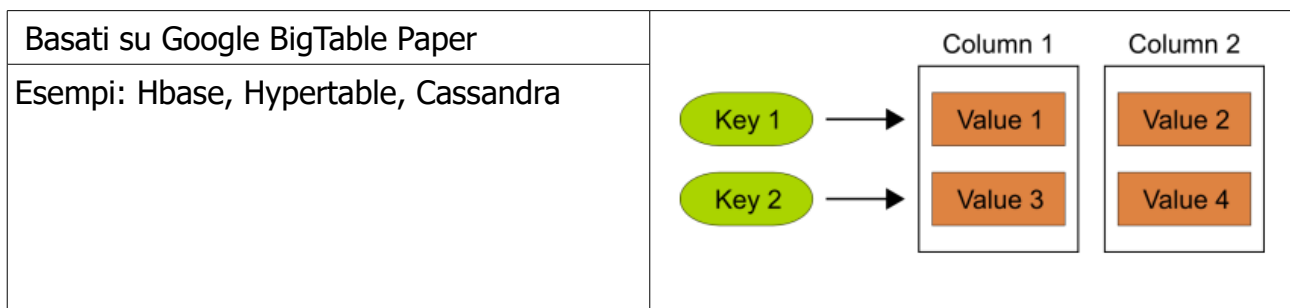
UI : 2 *nome → Sara | cognome → Piccolo | genere → Femminile | peso ->55*

L'aggiunta o l'eliminazione di una colonna non implica una ridefinizione dello schema (alter table).

Risparmio in termini di tempo e di memoria allocata (soprattutto considerando che le informazioni modificate spesso coinvolgono un numero ristretto di righe).



- ♣ Il punto di forza dei database colonnari riguarda l'immagazzinamento dei dati. Offrono un elevato grado di compressione dei dati e non sprecono spazio per campi vuoti.
- ♣ Utili con serie di dati storici o dati provenienti da più fonti (sensori, dispositivi mobile e siti web), quindi spesso differenti tra loro, e con elevata velocità.
- ♣ Indicati per sistemi di datawarehousing e read/only reporting, principalmente OLAP (On Line Analytical Processing). Sono infatti adatti all'uso di tecniche software per l'analisi interattiva e veloce di grandi quantità di dati.
- ♣ Modello di Dati: Big Table, Column Family.
- ♣ Pro: Ogni riga può avere un numero diverso di colonne, evitando la presenza di dati NULL.
- ♣ Pro: Una query recupera solo valori da determinate colonne e non dall'intera riga.
- ♣ Pro: Colonne composte da dati uniformi. Facilità di compressione e maggiore velocità di esecuzione e memorizzazione (possibilità di gestire PB).



## HBase

HBase è uno dei database column oriented la cui realizzazione ha preso spunto da Big Table. HBase è un progetto Apache e fa parte dell'ecosistema Hadoop.

L'integrazione con Hadoop è quindi molto elevata tant'è che il database fa uso di HDFS per la persistenza dei dati (così come Big Table usa GFS).

Il modello dati di HBase comprende il concetto di tabella e di column family proprio come BigTable. Una tabella può contenere una o più column family, che a loro volta possono contenere una o più colonne. In una tabella, ciascuna riga è contrassegnata da un rowID, mentre ciascun valore è contenuto in una cella identificata dalle coordinate (row, column, version), dove version è un timestamp.

Se nulla è specificato al momento di estrarre i valori dalla tabella, il valore più recente di ciascuna colonna è recuperato. I nomi delle colonne sono composti secondo la sintassi nome\_column\_family : nome\_colonna (per esempio anagrafica\_clienti:nome, anagrafica\_clienti:cognome). Le righe possono contenere valori per tutte le colonne o soltanto per alcune. Sul modello dati sono possibili le seguenti operazioni :

- » *Get* estrazione dei valori di una singola riga.
- » *Scan* estrazione dei valori di più righe.
- » *Put* inserimento di una nuova riga (o aggiornamento di una riga esistente).
- » *Delete* : cancellazione di una riga

Hbase può essere utilizzato anche attraverso HIVE, che è l'infrastruttura di data warehousing del framework Hadoop. La sua caratteristica fondamentale è l'utilizzo di un linguaggio molto simile ad SQL per eseguire operazioni di creazione, alimentazione ed

interrogazione di tabelle. Hive evita la scrittura di job MapReduce ed è un linguaggio adatto all'analista di business che non ha approfondite conoscenze informatiche. L'integrazione di Hive ed Hbase è realizzata attraverso la classe `org.apache.hadoop.hive.hbase.HBaseStorageHandler` e consente sia l'utilizzo di tabelle già esistenti in Hbase, sia la creazione di nuove direttamente da Hive.

### Cassandra.

Apache Cassandra è uno dei database column-oriented più diffusi e utilizzati anche da aziende molto famose quali Twitter ed eBay. Cassandra è un sistema distribuito fault tolerant, eventually consistent (quindi si rifà al CAP Theorem), e che possiede le caratteristiche di scalabilità lineare: le performance di lettura e di scrittura aumentano linearmente tramite l'aggiunta di nuovi nodi al cluster.

Secondo il teorema CAP, in un sistema distribuito è possibile garantire contemporaneamente soltanto due delle tre seguenti proprietà: Consistency (*tutti i nodi del sistema vedono gli stessi dati allo stesso istante temporale*), Availability (*ogni richiesta riceve una risposta sul buon fine o sull'insuccesso dell'operazione*) e Partition tolerance (*il sistema continua ad essere operativo malgrado la perdita di messaggi tra i nodi o la perdita di una parte del sistema stesso*).

Cassandra è stato sviluppato per rispettare le proprietà di availability e partition-tolerance, riuscendo così ad ottenere performance di tutto rispetto. Cassandra è un sistema eventually consistent e quindi alla fine raggiunge la consistenza.

Il modello dati prevede un'organizzazione che parte dal cluster come entità di più alto livello e scende fino alla singola colonna.

## Document-oriented DB

I database di tipo document-oriented sono progettati per memorizzare, ricercare e gestire dati semistrutturati (da non confondere con i sistemi di gestione documentale).

I "documenti", sono insiemi di coppie chiave/valore spesso organizzati in formato JSON o XML (formati autodescrittivi). Sono quindi una sorta di sofisticazione dei key/value DB.

La struttura però, non pone vincoli allo schema dei documenti, garantendone una grande flessibilità.

I dati non sono memorizzati in tabelle con campi uniformi e predefiniti, ma ogni documento è caratterizzato da specifiche caratteristiche (coppie chiave/valore strutturate in modo semplice)

```
{
  Nome:"Gianni",
  Indirizzo:"Via Roma 11",
  Hobby:"Fotografia"
}
```

Minore rigidità, non c'è uno schema standard. E' possibile aggiungere campi o ampliarli con dati (o parti di essi) multipli, inoltre non compaiono campi vuoti.

La struttura non pone dei vincoli allo schema e garantisce una buona flessibilità.

```
{
  Nome:"Pino",
  Indirizzo:"Via Galilei 34",
  Figli: [
    {Nome:"Luca", Eta:5},
    {Nome:"Anna", Eta:3},
  ]
}
```

Questi DB sono perfettamente adatti alla programmazione Object Oriented (OO Programming). Gli oggetti infatti possono essere serializzati in un documento eliminando il problema dell' *impedance mismatching*.

Sono particolarmente adatti quando i dati sono difficilmente rappresentabili con un modello relazionale per via dell'elevata complessità (dati complessi ed eterogenei).

>> Convenienti anche per rispondere alla Varietà dei Big Data, quando i dati hanno una struttura dinamica, o strutture molto differenti tra loro, o presentano un grande numero di dati opzionali.

>> Potrebbero essere utilizzati per implementare sistemi che formano uno strato su un DB relazionale o a oggetti.

>> Utilizzati particolarmente nel caso di flussi di dati medici o dati provenienti dai social network.

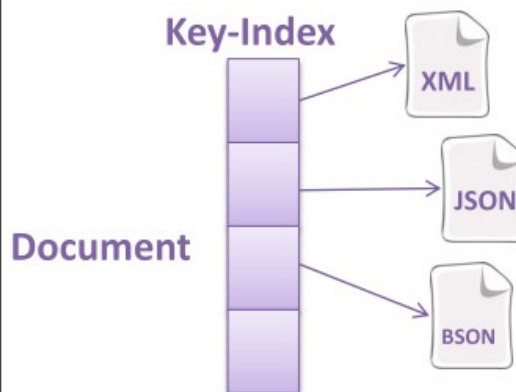
>> Ogni documento è individuato mediante una chiave univoca (stringa, URI o path). Di solito c'è un indice delle chiavi per velocizzare le ricerche.

>> Spesso sono presenti delle API o uno specifico linguaggio di interrogazione (query), per realizzare il recupero delle informazioni sulla base del contenuto, esempio il valore di uno specifico campo.

Il fine ultimo di questa tipologia di database non è quello di garantire un'alta concorrenza, ma quello di permettere la memorizzazione di grandi quantità di dati e di fornire un sistema di interrogazione sui dati performante.

Sono diventati i più popolari nella tipologia di NoSQL Database.

- Modello di Dati: raccolta di collezioni K-V (chiave-valore).
- Utilizzano le codifiche XML, YAML, JSON e BSON.
- Ispirati da Lotus Note (IBM).
- Esempi: **CouchDB, MongoDB, SimpleDB**



# MongoDB

E' uno strumento sviluppato in C++ in base ai principi tipici del movimento NoSQL, tra le sue principali caratteristiche abbiamo :

- La possibilità di gestire dati complessi
- Privilegiare le performance rispetto alle funzionalità fornite
- Portabilità, ovvero la possibilità di eseguire MondoDB su molteplici sistemi operativi
- Alta disponibilità attraverso le repliche
- Scalabilità attraverso lo sharing

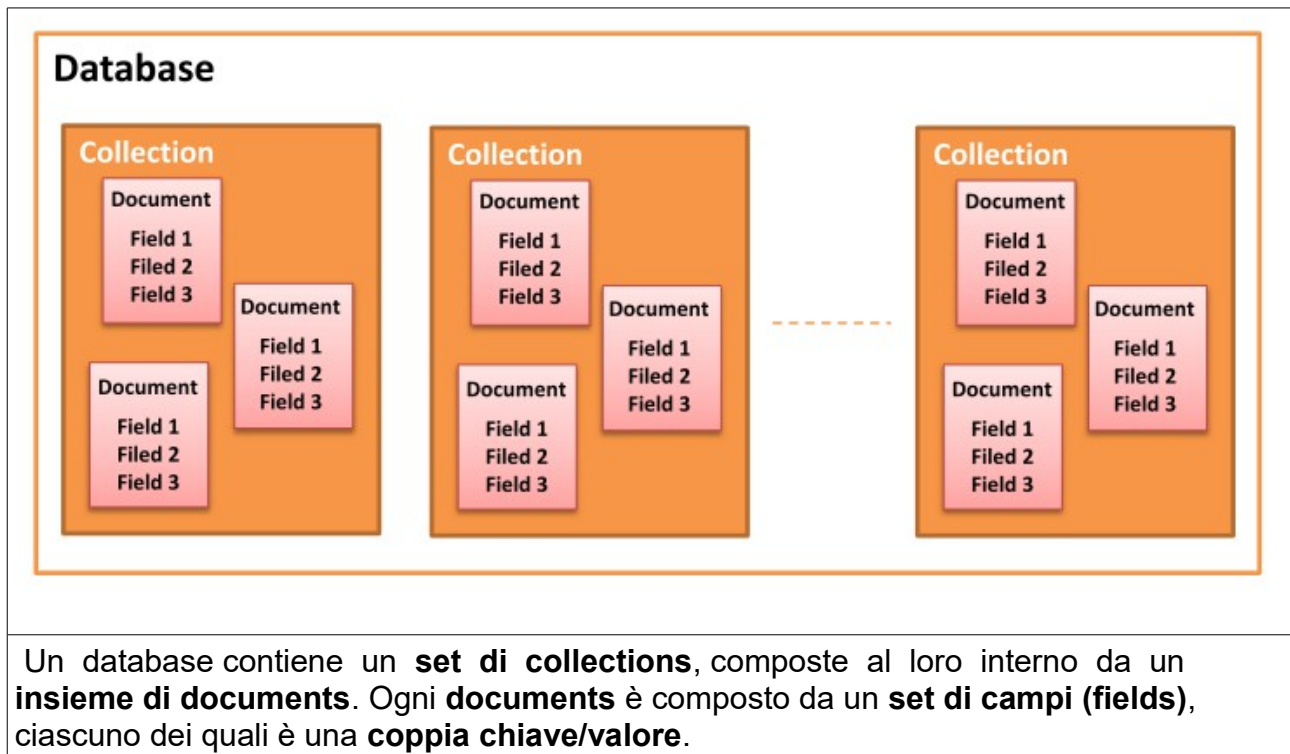
E' di tipo Document-oriented Database con schemi dinamici (schemaless). I dati archiviati sotto forma di document in stile JSON (BSON). BSON è una rappresentazione binaria simile a JSON, facilmente mappabile sia su oggetti JSON che su strutture tipiche di linguaggi OOP.

<p>JSON</p> <p>- Acronimo di JavaScript Object Notation, è un formato adatto per lo scambio dei dati in applicazioni client-server.</p>	<pre>{ "employees": [   { "firstName": "John", "lastName": "Doe" },   { "firstName": "Anna", "lastName": "Smith" },   { "firstName": "Peter", "lastName": "Jones" } ]}</pre>
<p>BSON</p> <p>- Binary JSON, è un'estensione del formato JSON che introduce anche il supporto di tipi di dati aggiuntivi: double, string, array, binary, null, date, int32, int64, object id...</p> <div data-bbox="172 1288 1417 1572"><pre>{ "hello": "world" } → "\x16\x00\x00\x00\x02hello\x00\x06\x00\x00\x00world\x00\x00" {"BSON": ["awesome", 5.05, 1986]} → "\x31\x00\x00\x00\x04BSON\x00\x26\x00\x00\x00\x020\x00\x08\x00\x00\x00awesome\x00\x011\x00\x33\x33\x33\x33\x33\x33\x14\x40\x102\x00\xc2\x07\x00\x00\x00\x00"</pre></div>	

I documenti di MongoDB possono contenere coppie chiave/valore, dove il valore può, a sua volta, contenere un altro documento o un array di documenti, oltre ai tipi di dato base (come string, date, double e così via). Ogni documento ha un campo predefinito, il campo `_id`, che può essere assegnato in fase di inserimento. In mancanza di un valore, il sistema ne assegna uno in modo univoco. I documenti BSON possono avere una dimensione massima di 16 MB, a meno di non utilizzare GridFS, una funzionalità di MongoDB, che salva i dati suddividendoli in parti (chunk) e che permette di superare quindi tale limite. Gli oggetti BSON sono raggruppabili in collection. Anche se sono assimilabili al concetto di Tabelle degli RDBMS, le collection non richiedono uno schema fisso : gli oggetti BSON che vi appartengono, non devono avere necessariamente lo stesso insieme di campi. Le collection a loro volta sono contenute in un Database.

Dalla versione 2.4.0 MongoDB ha il supporto ai dati geospaziali, attraverso il formato Geo JSON. Prima di questa versione tali dati potevano essere salvati come coppie di valori che esprimevano le coordinate. Il supporto consiste nella presenza di operatori per determinare l'inclusione di coordinate all'interno di un area, l'intersezione di superfici o la vicinanza di punti.

Un sistema basato su MongoDB contiene un set di Database così strutturati:



E' possibile individuare una sorta di **analogia** tra i concetti dei **database relazionali** e quelli presenti in **MongoDB**.

SQL	Mongo	Differenza
Database	<i>Database</i>	
Tabella	<i>Collection</i>	Le collection non impongono restrizioni sugli attributi (schemaless)
Riga	<i>Document</i>	Maggior supporto ai tipi di dato
Indici	<i>Indici</i>	Alcuni indici caratteristici
Join	<i>Embedding/linking</i>	Gestione delle referenze lato applicazione
Primary Key	<i>Object ID</i>	

La chiave è una stringa.

Il valore di un field può essere un tipo base (string, double, date...), oppure essere a sua volta un document o un array di document.

Ciascun documento ha il campo predefinito "\_id", che può essere settato in fase di inserimento, altrimenti il sistema ne assegnerà uno univoco in modo automatico.

```

{
  _id: ObjectId ('243252351224df35435bd35wre3'), //ID del documento
  nominativo: { nome: 'Gino', cognome: 'Bianchi'}, //valore di tipo documento
  data_nascita: new Date('Feb 12, 1975'),
  PIVA: '112334797634', //valori con tipo di dato base
  promozioni_utilizzate: ['PROMO 1', 'PROMO4'], //array
}

```

## MongoDB – Modellazione dei dati

La modellazione dei dati in MongoDB tiene conto in particolare di tematiche che riguardano:

- >> L'indicizzazione
- >> La normalizzazione o denormalizzazione dei dati
- >> Lo Sharding

♣ MongoDB ha uno schema flessibile. I documenti non hanno vincoli di struttura imposti dalle collection, infatti:

1. Stessa collection -> documenti con differenti campi o tipi di dato.
2. Un campo di un documento può contenere una struttura complessa come un array di valori o un altro documento.

In base alla seconda caratteristica è possibile:

♣ **Denormalizzare la struttura dati.** Tutti i dati che descrivono un'entità e quelli ad essi collegati sono inclusi all'interno del documento.

♣ **Si mantengono i documenti separati**, aggiungendo però ad ognuno di essi un riferimento che permetta di legarli, es. il campo "\_id".

### Struttura Denormalizzata

```

1 db.autori.insert ({
2   "_id": 3,
3   "Nome": "Gabriele",
4   "Cognome": "D'Annunzio",
5   "opere": [
6     { "titolo": "il piacere",
7       "anno": "1889",
8     },
9     { "titolo": "il fuoco",
10      "anno": "1900",
11     },
12     { "titolo": "la figlia di Iorio",
13       "anno": "1903",
14     }
15   ]
16 })

```



## Struttura Normalizzata

```

1 db.otori.insert({
2   "_id": 3,
3   "Nome": "Gabriele",
4   "Cognome": "D'Annunzio",
5 })
6 db.libri.insert({
7   "titolo": "il piacere",
8   "anno": "1889",
9   "id_autore": 3
10 })
11 db.libri.insert({
12   "titolo": "il fuoco",
13   "anno": "1900",
14   "id_autore": 3
15 })
16 db.libri.insert({
17   "titolo": "la figlia di Iorio",
18   "anno": "1903",
19   "id_autore": 3
20 })

```

- ♣ **MongoDB non supporta le JOIN.** Se si adotta la struttura normalizzata, le **relazioni** tra i documenti si esplicitano tramite **query in cascata** che recuperano i documenti.
- ♣ **Convenzione DBRef** – specifica un **documento** tramite: **database -> collection -> \_id**
- ♣ Da **riga di comando** è possibile eseguire delle **semplici istruzioni**, oppure lanciare **script** salvati su file.

Istruzione	Esempio	Descrizione
<b>db</b>	db	Restituisce il database corrente.
<b>use &lt;nome db&gt;</b>	use impiegati	Cambia il db corrente in impiegati.
<b>db.&lt;collection&gt;.insert()</b>	db.studenti.insert({ nome: 'Gianni', Cognome: 'Verdi'})	Inserisce un utente nella collection studenti senza specificare l'ID
<b>db.&lt;collection&gt;.find()</b>	dc.studenti.find()	Elenca tutti i documenti della collection studenti
<b>db.&lt;collection&gt;.drop()</b>	db.studenti.drop()	Elimina l'intera collection dal DB

Oltre ai comandi per effettuare **inserimenti**, **modifiche** e **recupero** di documenti, esistono anche i **cursori**, cioè degli oggetti che permettono di **iterare su un set di documenti** restituiti come **risultato di una query**.

```

1  import com.mongodb.MongoClient;
2  import com.mongodb.DB;
3  import com.mongodb.DBCollection;
4  import com.mongodb.BasicDBObject;
5  import com.mongodb.DBCursor;
6
7  public class MongoDB_test {
8      public static void main(String args[]) throws Exception {
9          //istanza studente
10         MongoClient mongoClient = new MongoClient();
11         //recuperiamo il database
12         DB db = mongoClient.getDB("test");
13         //creiamo una collection
14         DBCollection coll = db.createCollection("studenti", null);
15         //creiamo un documento
16         BasicDBObject doc =
17             new BasicDBObject("nome", "Gianni").
18             append("cognome", "Verdi").
19             append("indirizzo", new BasicDBObject("comune", "Firenze")).
20             append("provincia", "FI");
21         //aggiungiamo il documento alla collection
22         coll.insert(doc);
23         //creiamo un documento per l'esecuzione di una query
24         BasicDBObject query = new BasicDBObject("cognome", "Verdi");
25         //utilizziamo un cursore per iterare sui risultati della
26         //query e mostrare ogni documento
27         DBCursor cursor = coll.find(query);
28         try {
29             while(cursor.hasNext()){
30                 System.out.println(cursor.next());
31             }
32         } finally {
33             cursor.close();
34         }
35         System.out.println("");
36     }
37 }

```

Nell' esempio con l'uso delle **API Java** si:

- **Creano le istanze del client *MongoDB*** e del **database *test***.
- **Crea la nuova collection "*studenti*".**
- **Crea un documento** che modella uno studente, con i campi ***nome, cognome e indirizzo*** (documento annidato).
- **Inserisce** il documento nella collection.
- **Crea e si esegue la query** che effettua una ricerca sul campo cognome.
- **Usa un *cursore*** per stampare a video i documenti recuperati dalla query.

## MongoDB – Esempi di Query

### 1. **Selezione di tutti i documenti in una collezione**

```
db.inventario.find()
```

### 2. **Corrispondenza esatta su un Array (campo multivalore)**

```
db.inventario.find({tags:['frutta','carne','pesce']})
```

### 3. **Specifica condizione AND**

```
db.inventario.find({type:'carne',prezzo:{$lt:9.95}})
```

### 4. **Corrispondenza su un campo in un Subdocument**

```
db.inventario.find({'producer.company': 'ABC123'})
```

MongoDB ha una funzione integrata che consente di scrivere job in stile MapReduce. Si tratta di una implementazione interne a MongoDB, che non interagisce con l'ambiente Hadoop, tuttavia si tratta di uno strumento di calcolo piuttosto potente, di cui è interessante conoscerne il funzionamento.

Esiste anche un adapter Hadoop per MongoDB e questo permette la comunicazione tra i due ambienti, consentendo a MongoDB di essere utilizzato come fonte dati sia come destinazione per i job MapReduce di Hadoop. L'adapter mette a disposizione alcuni tipi dato compatibili con MongoDB (BSONObject e BSONObjectWrutable), utilizzabili nella scrittura delle classi Mapper e Reducer. In aggiunta a esse, occorre implementare una classe, che derivi da MongoTool (inclusa nell'adapter), il cui compito consiste nel leggere i file di configurazione XML. In essi sono presenti tutti i parametri necessari al funzionamento dei job MapReduce in combinazione con MongoDB, come per esempio, l'URI del server, il nome del database e delle collection di input e di output, oppure le classi da utilizzare come InputFormat e OutputFormat.

La combinazione Hadoop-MongoDB può essere utile in scenari in cui sia necessario eseguire grandi elaborazioni batch, salvandone il risultato di nuovo su MongoDB. Un altro scenario di utilizzo potrebbe consistere nell'estrazione dei dati da MongoDB e nell'elaborazione in Hadoop al fine di prepararli per essere inseriti in un sistema di data warehousing e analisi esterno.

## Graph Database

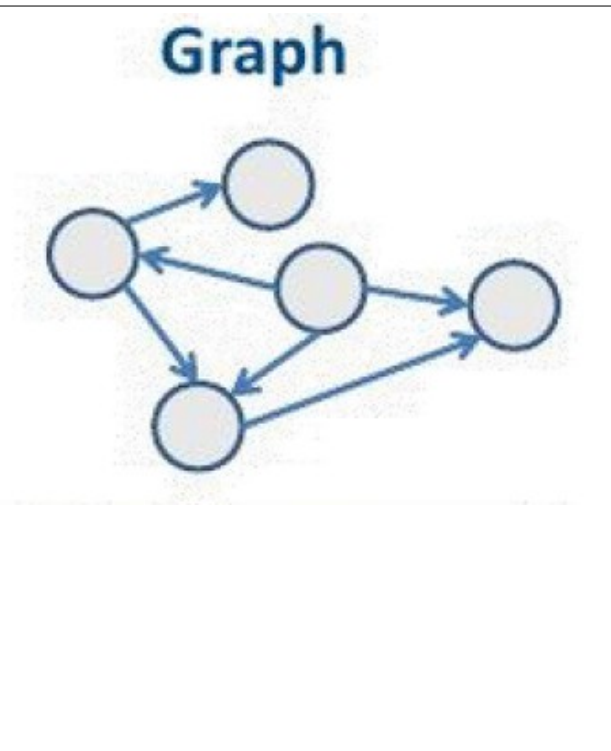
Nascono dall'esigenza di dover gestire dati fortemente connessi tra loro (specie non tabulari), in cui il concetto di relazione tra i dati ha un elevato potenziale informativo.

Il concetto matematico di grafo consiste in un insieme di elementi detti *nodi o vertici* / *vertex*) collegati fra loro da *archi o lati* (*edge*). I grafi rivestono un grande interesse per un ampio spettro di applicazioni e sono utilizzati in numerosi campi della matematica. In informatica il grafo indica una struttura dati costituita da un insieme finito di coppie ordinate (gli archi) di oggetti (i nodi) che possono essere parte della struttura del grafo, oppure possono essere riferimenti a oggetti esterni. Inoltre la struttura può avere anche valori associati agli archi, come ad esempio dati numerici o stringhe (quantità, costi, nomi etc..). Alcune strutture dati che sono utilizzate spesso nella programmazione, come map, tree, list o il concetto di oggetto, sono in realtà casi particolari di grafi.

Occupano meno spazio rispetto al volume di dati con cui sono fatti e memorizzano molte informazioni sulla relazione tra i dati.

Questi DB si prestano molto bene alla rappresentazione di dati semistruzzurati e interconnessi **come pagine Web, hiperlink e Social Network, reti di computer.**

*Le relazioni complesse presenti nei Social Network in realtà sarebbero rappresentabili anche con i **DB NoSQL** di tipo key/value, column-oriented e document oriented. Pur non esistendo il concetto di relazione, si tratterebbe di salvare in ciascuna entità una serie di riferimenti (per esempio chiavi o row key) ad altre entità. Una base dati così costituita sarebbe complessa sia da interrogare che da aggiornare a fronte di cambiamenti. In modo simile avremo dei problemi a rappresentare un grafo con un sistema RDBMS.*



Non esiste il concetto di relazione, ciò comporta una maggiore complessità nelle interrogazioni e negli aggiornamenti.

Graph DB sono ottimizzati alla rappresentazione e alla navigazione efficiente dei dati:

- ♣ Non hanno uno schema rigido.
- ♣ Si adattano a cambiamenti nelle strutture dei dati.
- ♣ Svolgono facilmente operazioni sui collegamenti, come ad esempio individuazione di "amici degli amici", oppure il calcolo del percorso più breve tra due nodi.

Uno dei modelli maggiormente utilizzati per **implementare i graph database** è quello basato sui **“property graph”**.

♣ **I nodi**

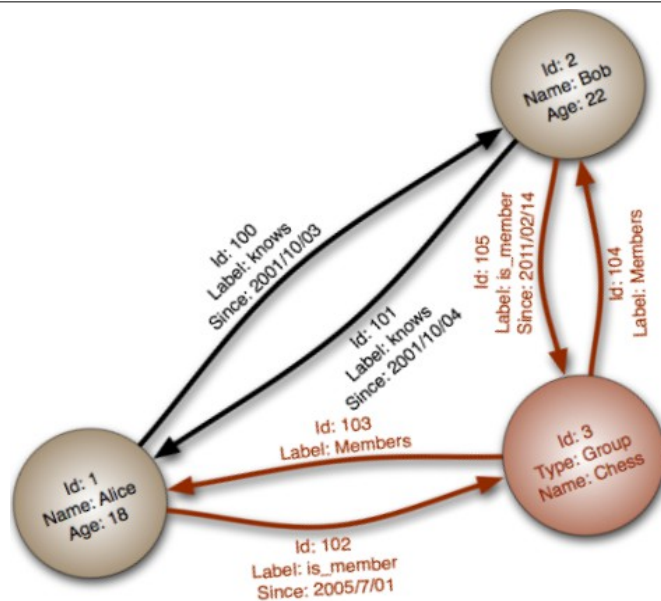
cioè dei contenitori di proprietà.

♣ **Le proprietà**

esprese sotto forma di chiave valore.

♣ **Le relazioni**

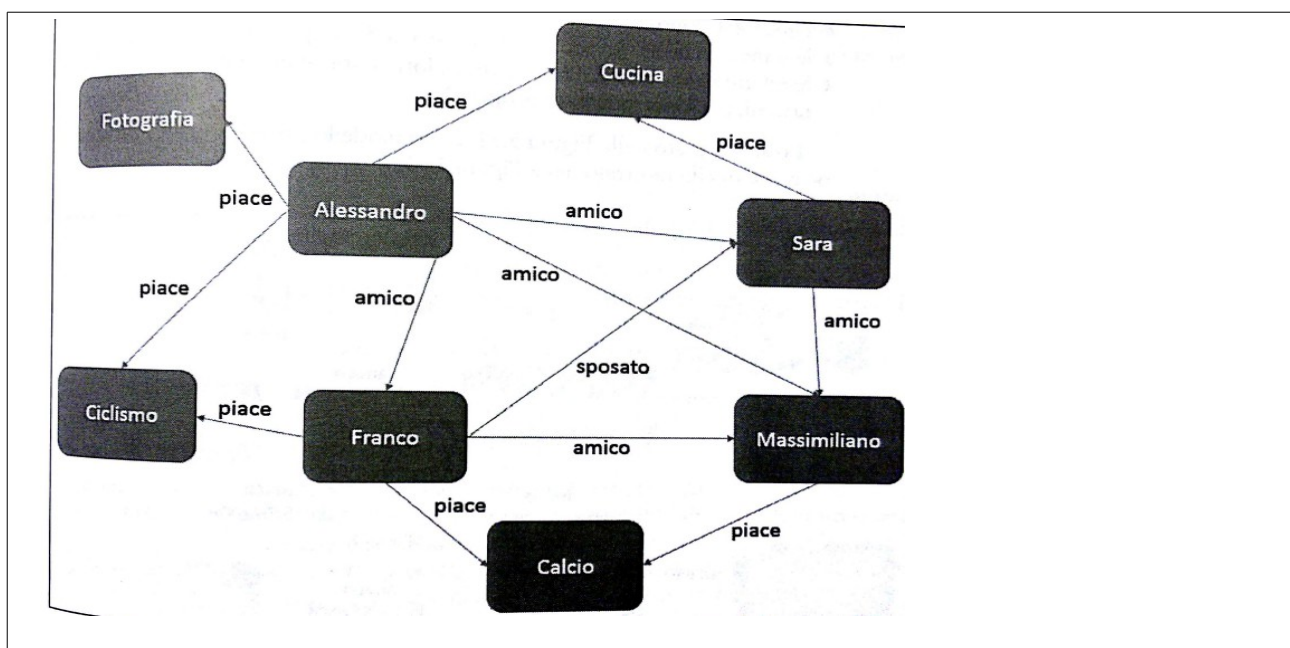
legano i nodi e a volte possono contenere proprietà.



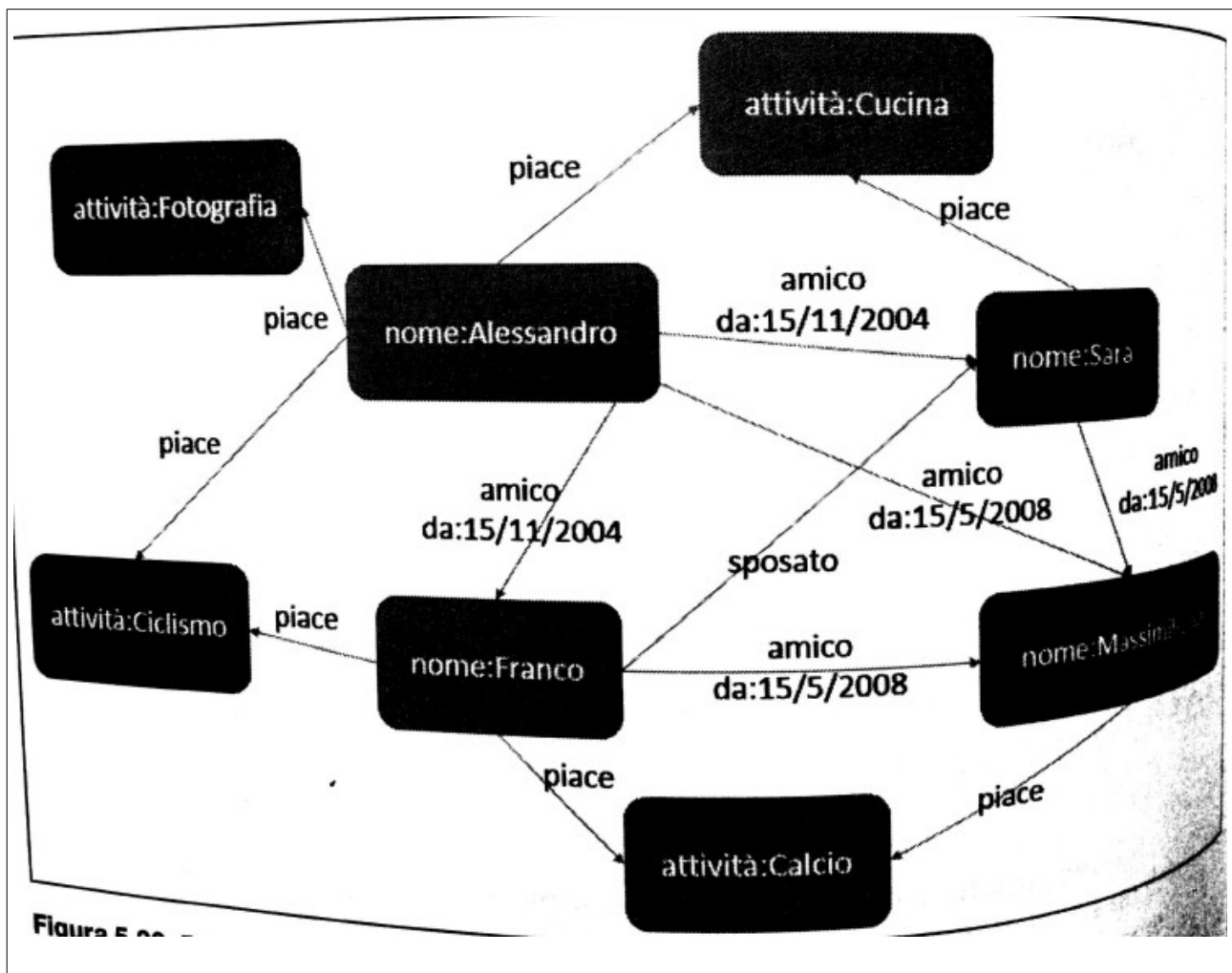
Nella figura si vede la rappresentazione a grafo con le relazioni tra i nodi ed i valori associati ad entrambe le entità.

- ♣ Modello di Dati: nodi-relazioni (chiave/valore).
- ♣ Adatti a gestire dati specifici o che cambiano seguendo un certo schema.
- ♣ Utilizzati per la gestione di dati geospaziali, analisi di reti, bioinformatica, Social Network e motori di raccomandazioni, dove i dati sono caratterizzati da interconnessioni.
- ♣ Basati sulla Teoria dei Grafi.
- ♣ Esempi: Neo4J, AllegroGraph, Objectivity.
- ♣ (-) Non è facile eseguire query su questo tipo di database.

*Esempio di una struttura a grafo :*



Il modello del grafo precedente con il property graph, potrebbe diventare così :



Un utilizzo che può riguardare in maniera trasversale numerosi settori di attività, è il "master data management" : molto spesso i master data contengono gerarchie e relazioni non molto ben definite e magari variabili nel tempo, difficili quindi da rappresentare attraverso un RDBMS, che ha una struttura rigida e definita a priori.

Il Master Data Management (MDM) indica l'insieme degli strumenti tecnologici e dei processi utilizzati per creare e mantenere in modo consistente ed accurato i cosiddetti master data.

I Master Data rappresentano entità importanti per il business (cliente, prodotto...) e sono utilizzati in numerosi sistemi all'interno dell'azienda; inoltre riguardano generalmente entità complesse (con molti attributi descrittivi) ed entità con una cardinalità elevata.

Esistono numerose implementazioni di graph database. Tra i più utilizzati vi sono : Neo4j, AllegroGraph (un database proprietario) e Titan (un graph database open source). E' da segnalare anche un'iniziativa di Microsoft Research per la creazione di graph database chiamato Trinity.

Neo4j è un graph database open source molto performante, basato sul modello property graph. E' scaricabile in tre versioni : la *community edition*, non prevede né il monitoring disponibile nella versione *advanced* né l'alta disponibilità attraverso il clustering presente



nella versione *enterprise*. Queste ultime sono a pagamento soltanto se non sono utilizzate in un progetto open source.

Le caratteristiche principali di Neo4J sono le seguenti :

Sono garantite le proprietà ACID delle transazioni

Alta scalabilità (nell'ordine dei miliardi di nodi del grafo)

Ottime performance nell'attraversamento dei nodi

L'accesso al database avviene in tre modi diversi : l'API Java, la console da riga di comando e l'interfaccia web. Quest'ultima consente di effettuare attività di monitoraggio tramite la dashboard o la sezione *Server Info*. Mette a disposizione un accesso alla console e permette la navigazione o la modifica dei dati tramite il Data browser.

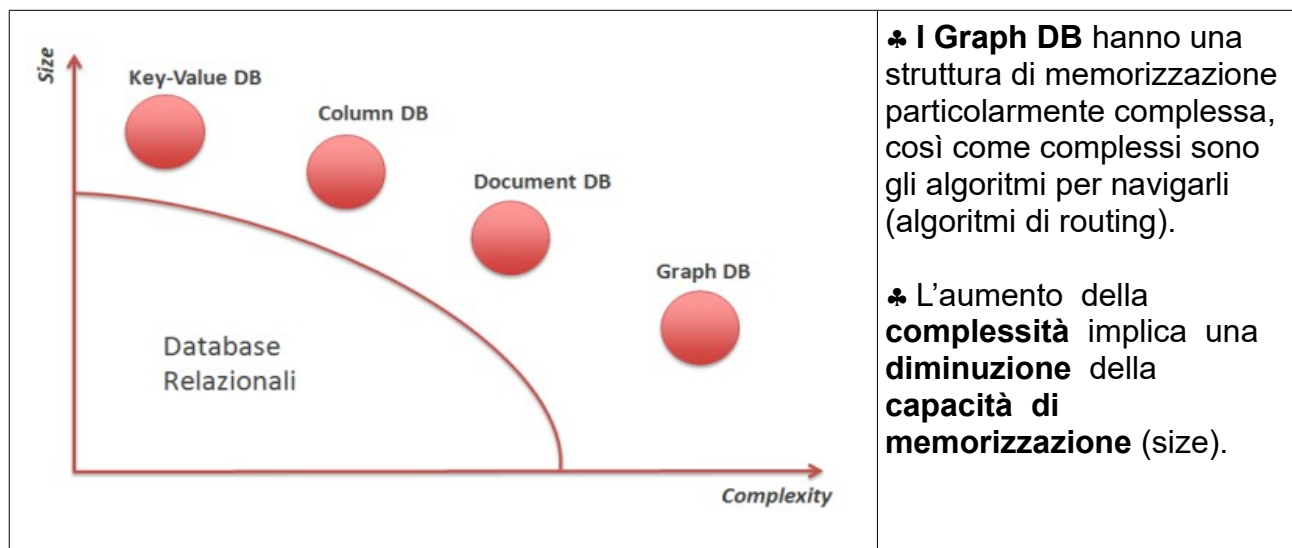
Il Data browser funziona attraverso la query Cypher, il linguaggio di interrogazione di Neo4J, e ritorna una lista di nodi o relazioni, oppure una visualizzazione grafica degli stessi. Sempre attraverso il Data browser è possibile inserire o eliminare nodi e relazioni, oppure creare proprietà da associare al nodo o alla relazione. Cypher è un linguaggio dichiarativo pensato per facilitare l'accesso ai nodi, il loro attraversamento ed eventualmente le loro modifica. La query Cypher, oltre che dal Data browser, possono essere eseguite dal tool da riga di comando, da API HTTP REST oppure dalle API Java.

## Comparazione

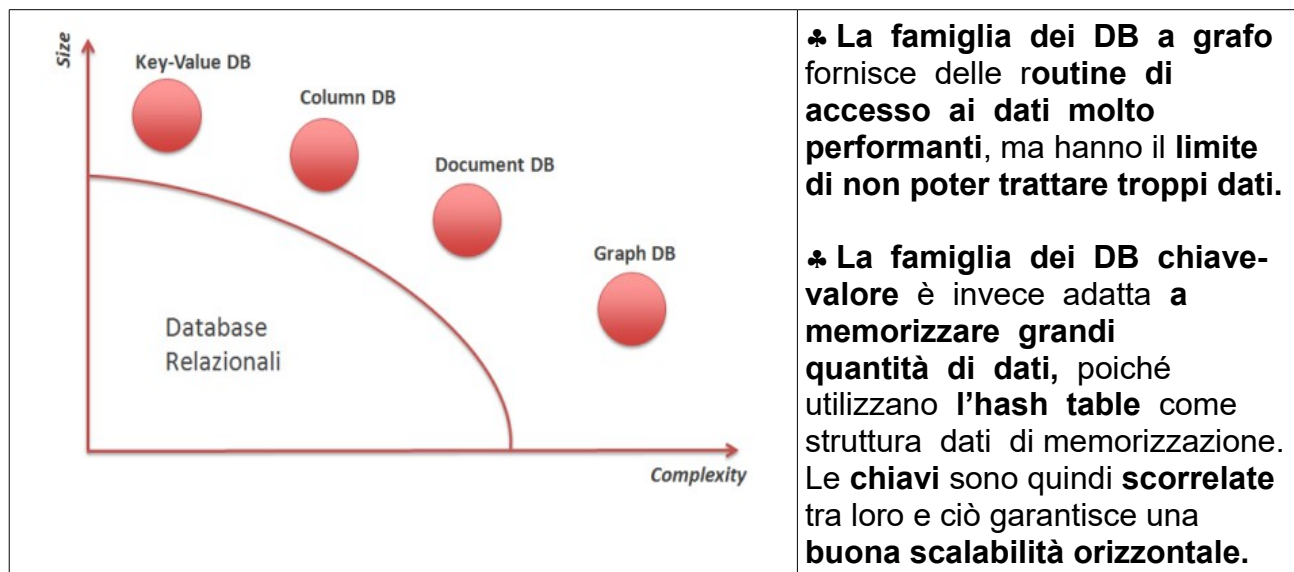
Tabella di riepilogo delle caratteristiche principali delle quattro classi fondamentali di NoSQL Database.

	Key-Value	Document-Oriented	Column-Family	Graph-Oriented
Schema	NO	NO	SI	NO
Correlazione dati	NO	NO	SI	SI
Caratteristica principale	Altissima Concorrenza	Query performanti, moli di dati altissime	Singola riga CF non separabile	Presenza delle relazioni tra i dati

Valutazione delle soluzioni NoSQL DB sulla base dei parametri **size** e **operational complexity**.



Valutazione delle soluzioni NoSQL DB sulla base dei parametri **size** e **operational complexity**.



Queste sono le principali famiglie di database NoSQL che ho preso in esame. Per completezza cito anche i seguenti DB NoSQL :

- ♣ XML DB
- ♣ Object DB
- ♣ Multivalued DB
- ♣ ACID NoSQL

## XML Database

I Database XML sono nati come ulteriore soluzione alla gestione di dati difficilmente rappresentabili in formato tabellare, infatti:

```
<?xml version="1.0"?>
<quiz>
  <qanda seq="1">
    <question>
      Who was the forty-second
      president of the U.S.A.?
    </question>
    <answer>
      William Jefferson Clinton
    </answer>
  </qanda>
  <!-- Note: We need to add
  more questions later.-->
</quiz>
```

**XML**

- ♣ Il formato XML è molto utilizzato per lo scambio dei dati; e la vista XML è quella preferita da utenti e applicazioni.
- ♣ I documenti XML si prestano bene a contenere strutture dati gerarchiche.
- ♣ Nasce quindi l'esigenza di gestire dati XML all'interno dei database tenendo ben presente la struttura di origine.

Gli XML-DB organizzano i dati in documenti XML ottimizzati per contenere una grande quantità di dati semi-strutturati.

In questo contesto esistono tre possibili soluzioni:

- ♣ Middleware  
Strati software che permettono lo scambio di dati XML tra database(tradizionali) e applicazioni.
- ♣ Database con supporto XML (o XML-enabled)
- ♣ Database nativi XML

Inizialmente lo sviluppo middleware ha permesso di far dialogare i due mondi. Poi sono stati sviluppati diversi XML DBMS nativi, e attualmente diversi DBMS commerciali stanno investendo nel supporto XML.

♣ Database con supporto XML (Oracle 9i con XDB) permettono di gestire documenti XML in 2 modalità:

1. Document-Centric

♣ I documenti XML sono i dati da inserire nel DB (come campi).

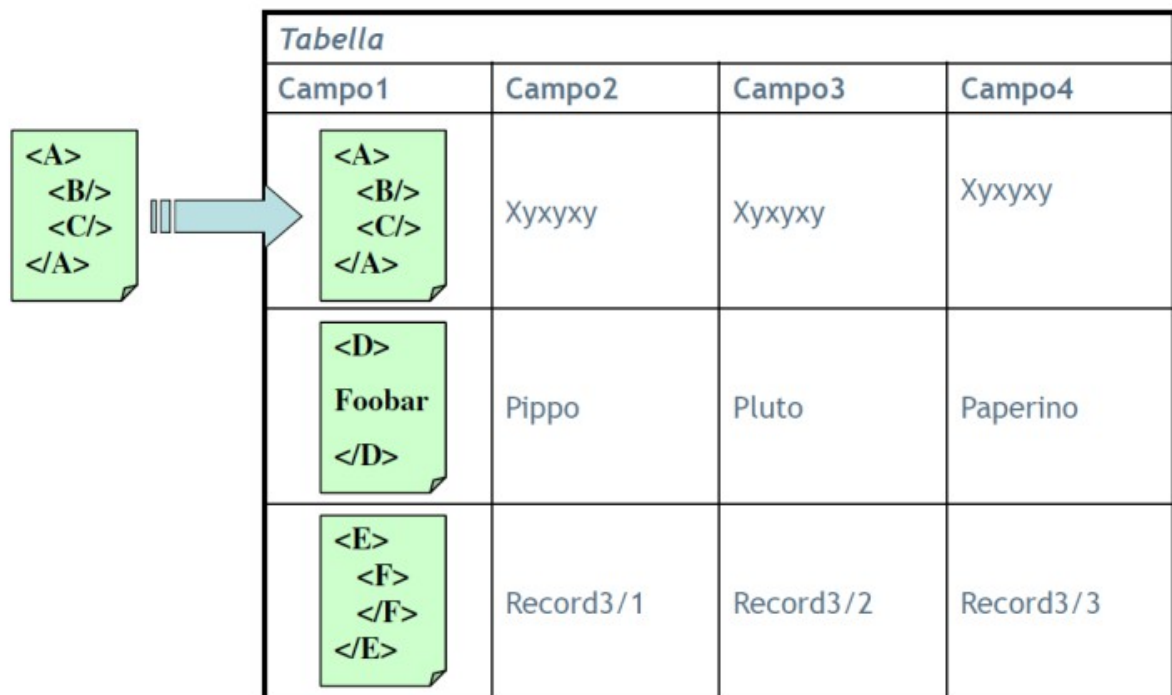
♣ Su questi documenti possono essere eseguite semplici query tramite XPath o espressioni regolari.

2. Data-Centric

♣ La struttura gerarchica del documento XML è parte fondamentale dei dati, e viene mappata su un insieme di tabelle relazionali.

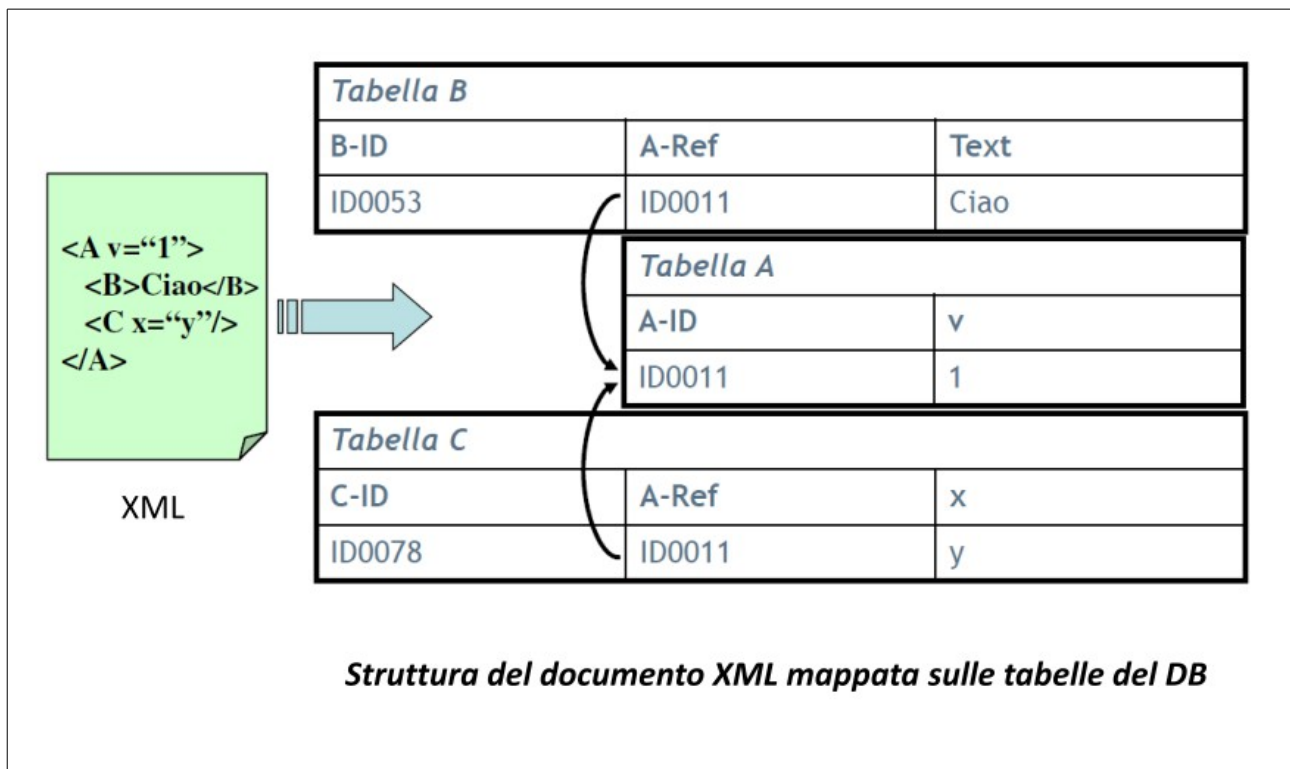
♣ Dalle relazioni tra le tabelle è possibile ricostruire la struttura originaria del documento XML.

**Document Centric**



***Documento XML inserito nei campi del DB***

## Data Centric



Gli XML-DB organizzano i dati in documenti XML che possono essere:

- ♣ Facilmente interrogati tramite query XPath.
- ♣ Trasformati mediante XSLT in output tabellari.

### XPATH

- ♣ Linguaggio che consente di estrarre e manipolare nodi XML o valori.
- ♣ Consente la creazione di indici e query su dati di tipo XML, gestiti nei DBMS relazionali.

### XSLT (eXtensible Stylesheet Language Transformation)

- ♣ Linguaggio che permette la trasformazione di documenti XML in:
  - ♣ Altri documenti XML, ma con differente struttura.
  - ♣ In HTML o documenti di testo (TXT, CSV...)

♣ Xpath è un linguaggio in cui compaiono delle path expression, cioè una sequenza di passi (percorso) per raggiungere un nodo XML. In queste espressioni gli elementi sono separati dal carattere '/'.

- ♣ XPath presenta due possibili notazioni:

### Sintassi abbreviata

compatta consente la facile realizzazione di costrutti intuitivi.

	♣ Questa espressione seleziona gli elementi C che sono figli di elementi B a loro volta figli dell'elemento A (radice del file XML).
--	--

**A//B/\*[1]**

♣ Espressione che seleziona il primo elemento figlio di B, senza tener conto di quanti elementi intercorrono tra B e A, //. Questa volta A non è nodo radice.

### Sintassi completa

Più complessa, contiene maggiori opzioni per specificare gli elementi.

**/A/B/C**



**/child::A/child::B/child::C**

**A//B/\*[1]**



**child::A/descendant-or-self::B/child::node()[1]**

♣ In questo caso in ogni punto del Xpath viene specificato esplicitamente l'Axis seguito da ::

Axis

♣ L'Axis è l'elemento che indica il senso in cui deve essere percorso l'albero del documento XML.

♣ Alcuni Axis sono: child, attribute (@), descendant-or-self (//), parent(..), following etc.

### XML Database – Esempio XPath

**/listautenti/account//telefoni/\***

Restituisce la lista di tutti i nodi interni al nodo telefoni, cioè fisso, cellulare e fax.

**/listautenti/account//indirizzo/..**

Restituisce tutti i nodi che contengono un nodo indirizzo. L'uso dell'Axis // permette che vengano individuati anche nodi di livelli differenti purché presenti all'interno di account.

**string(descendant::nome[1])**

Restituisce il valore della stringa del primo elemento nome che trova, in questo caso Gianni R.

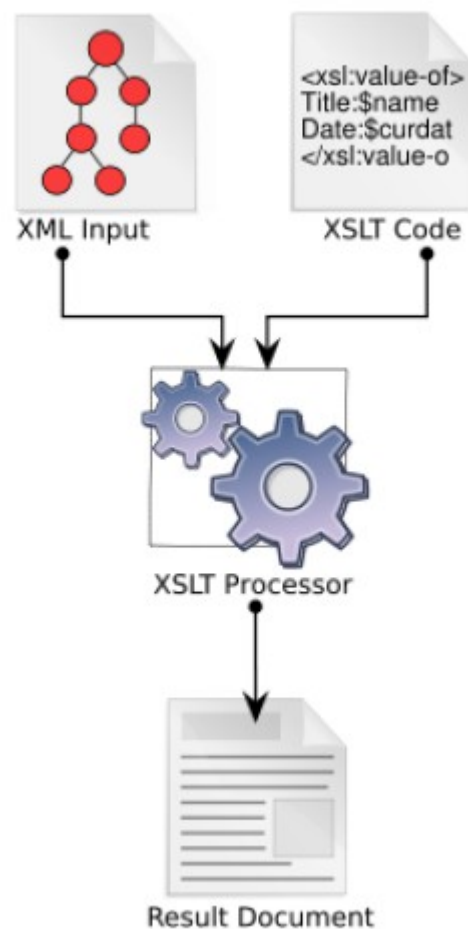
```
2 <?xml version="1.0" encoding="UTF-8"?>
3 <listautenti>
4   <account user="gianni">
5     <mail>gianni_red@mail.com</mail>
6     <nome>Gianni R.</nome>
7     <principale>
8       <indirizzo>
9         <via>Via vecchia 1</via>
10        <cap>98100</cap>
11        <citta>Siena</citta>
12      </indirizzo>
13      <telefoni>
14        <fisso>090123456</fisso>
15        <cellulare gestore="tim">3331214567</cellulare>
16      </telefoni>
17    </principale>
18    <altreirecapiti>
19      <ufficio>
20        <indirizzo>
21          <via>Via del lavoro, 2</via>
22          <cap>98100</cap>
23          <citta>Siena</citta>
24        </indirizzo>
25        <telefoni>
26          <fisso>09078901</fisso>
27          <fax>090345367</fax>
28        </telefoni>
29      </ufficio>
30    </altreirecapiti>
31  </account>
32 </listautenti>
```



XSLT (eXtensible Stylesheet Language Transformations) è il linguaggio per trasformare l'XML (diventato uno standard web con una direttiva W3C del 16 novembre 1999) in un altro documento.

♣ Per realizzare la trasformazione XSLT occorrono due file:

1. Il documento XML che deve essere trasformato.
2. Il foglio di stile XSLT che fornisce la semantica della trasformazione. Questo documento, infatti, interpreta un XML come una serie di nodi strutturati ad albero.



**Documento XML che deve essere trasformato.**

```
<?xml version="1.0" ?>
<persons>
  <person username="JS1">
    <name>John</name>
    <family-name>Smith</family-name>
  </person>
  <person username="MI1">
    <name>Morka</name>
    <family-name>Ismincius</family-name>
  </person>
</persons>
```

Lo si vuole trasformare in un nuovo documento XML con una **struttura differente**. Il foglio di stile XSLT definisce il template da seguire nel seguente modo:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:output method="xml" indent="yes" />

  <xsl:template match="/persons">
    <root>
      <xsl:apply-templates select="person" />
    </root>
  </xsl:template>

  <xsl:template match="person">
    <name username="{@username}">
      <xsl:value-of select="name" />
    </name>
  </xsl:template>

</xsl:stylesheet>
```

Il nuovo documento XML che si ottiene in output avrà la seguente struttura:

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
  <name username="JS1">John</name>
  <name username="MI1">Morka</name>
</root>
```

#### Osservazione

Un documento XML può essere associato a più fogli di stile XSLT, ciascuno dei quali genererà un output diverso. Lo stesso procedimento vale anche al contrario: uno stesso foglio di stile può essere applicato a più documenti XML, allo scopo di produrre documenti dal contenuto differente ma formato analogo.

- ♣ Database XML Nativi consentono la gestione di documenti XML come collezioni di dati indicizzati, che possono essere interrogati mediante XPath o Xquery.
- ♣ Possibilità di integrazione con DB relazionali standard, con schemi che mappano la struttura relazionale su documenti XML.
- ♣ Esempi: Tamino (SoftwareAG), BaseX, eXsist (Apache).
- ♣ Attualmente il progetto maggiormente sviluppato e stabile è eXistdb.

Supporta le interrogazioni tramite XPath e XQuery 1.0

Supporta gli aggiornamenti tramite il linguaggio Xupdate

Accessibile da codice mediante l'interfaccia standard XML:D

Il sito del movimento NoSQL (<http://nosql-database.org/>), attualmente indicizza più di 255 DB NoSQL, suddivisi in queste 14 categorie (in questa relazione sono stati illustrati 5 famiglie) :

**Wide Column Store / Column Families**

**Document Store**

**Key Value / Tuple Store**

**Graph Databases**

**Multimodel Databases**

**Object Databases**

**Grid & Cloud Database Solutions**

**XML Databases**

**Multidimensional Databases**

**Multivalued Databases**

**Event Sourcing**

**Other NoSQL related databases**

**Scientific and Specialized Dbs**

**unresolved and uncategorized**

## ***Bibliografia***

- 1) Big Data – *Architetture, tecnologie e metodi per l'utilizzo di grandi basi di dati* – Alessandro Rezzani – Apogeo, Maggioli Editore
- 2) Slides “BIG DATA” dell' Ing. Mariano Di Claudio – Lezione del 17/09/2014
- 3) Slides “BIG DATA” dell' Ing. Mariano Di Claudio – Lezione del 22/10/2014
- 4) NoSQL ... a view from the top Part 2 - Red Stack Tech Ltd James Anthony