mokabyte.it

Il teorema CAP... in Brewer - MokaByte

Luca Vetti Tagliati

25-33 minuti

Con questo quinto articolo concludiamo la serie dedicata al teorema CAP, più precisamente alle sue ripercussioni sulle architetture distribuite, illustrando le scelte operate dal team di MongoDB per far fronte all'impossibilità di fornire contemporaneamente le tre feature di totale Consistenza, continua disponibilità (Availability) e tolleranza alle Partizioni, come sancito dal teorema.

Introduzione

Nel corso del precedente articolo, dopo aver fornito le motivazioni storiche e una breve panoramica delle diverse categorie di database NoSQL (**Document**, **Graph Store**, **Key Value** e **Wide Column**) abbiamo iniziato l'esplorazione di **MongoDB** [1]. In particolare, abbiamo visto come MongoDB appartenga alla categoria **Document**: banche dati che associano ad ogni **chiave** una **struttura** di **dati complessa**, definita appunto documento. Un documento può contenere numerose coppie **chiave-valore** ("key-value pairs"), o coppie **chiave-array** o anche **documenti annidati**. In MongoDB, le strutture e i dati sono rappresentati per

mezzo di una codifica standard, denominata **BSON** (Binary JSON) in quanto ottimizzazione della codifica JSON. Questa organizzazione dei dati è definita **semistrutturata** proprio perche' non c'è separazione tra i dati e il corrispondente schema: la quantità di struttura definita dipende dallo specifico utilizzo.

Quale compromesso?

La classificazione comunemente accettata per MongoDB prevede una sua appartenenza al gruppo **CP** [2]: elevata **Coerenza** e tolleranza alle **Partizioni**. A questa categoria appartengono le soluzioni in cui i dati sono mantenuti in maniera coerente in tutti i nodi del cluster, e viene garantita la tolleranza alla partizione: ciò evita che dati presenti sui vari nodi possano desincronizzarsi, a spese però della disponibilità. Infatti il sistema può divenire non disponibile quando un nodo va giù.

In questo articolo presentiamo le scelte architetturali che portano a questo compromesso e vediamo come questo **non** sia il solo modo di operare di MongoDB.

Replica dei dati

Come visto in precedenza, i **sistemi distribuiti** utilizzano avanzate tecniche di distribuzione dei dati non solo al fine di poter rispondere alla crescente domanda di operare efficientemente su quantità di dati sempre più vaste, ma anche per fornire la necessaria **ridondanza** atta a migliorare feature come **disponibilità/scalabilità/fault tolerance**.

Potendo disporre di più copie dei medesimi dati distribuite su diversi nodi/sever è possibile proteggere il database dalla perdita dei singoli nodi/server. Le repliche inoltre permettono di

recuperare lo stato di funzionamento corretto a seguito di guasti hardware e/o interruzioni del servizio. Le varie repliche sono anche uno dei principali meccanismi utilizzati dai sistema distribuiti per supportare la scalabilità (in questo caso si parla di scalabilità **orizzontale**). Per esempio è possibile soddisfare un numero elevato di richieste di acquisizione di dati da parte dei sistemi clienti (o più semplicemente del numero stesso dei clienti) tramite l'inoltro delle richieste di lettura a server diversi.

È infine possibile avere nodi in centri dati **geograficamente dislocati** per migliorare le performance dei sistemi distribuiti sul globo: in questo caso i vari client possono accedere a nodi localizzati piuttosto che eseguire richieste a nodi ubicati in siti distanti (richieste over the WAN).

Cluster in MondoDB

Per quanto attiene a MongoDB, il classico **cluster** è tipicamente denominato "insieme di repliche" (**replica set**). Questo non è altro che un gruppo di daemon MongoDB in esecuzione (processi residenti/daemon), chiamati **mongod**. Ognuno di questi processi è in grado di gestire le richieste dei client, il formato di dati e di eseguire le varie operazioni di gestione del server in background. Un **insieme** di **repliche** è quindi l'**equivalente** del **cluster** Oracle Coherence.

La configurazione dell'insieme di repliche di **mongod** prevede per default la presenza di **un solo nodo primario** destinatario di tutte le richieste di **scrittura** dati. Tutti i restanti nodi, detti **secondari**, hanno la responsabilità di eseguire le **operazioni dettate** dal **nodo primario** al fine di mantenere consistente la base dati (figura 1). Si tratta quindi di una versione della più classica delle

architetture **master/slave**.

Figura 1 – Replica delle scritture in MongoDB.

Il **nodo primario**, quindi, gestisce tutte le operazioni di scrittura da parte dei client. Un solo nodo primario, tipicamente, prevede un numero di repliche, mentre **ogni replica** ha **un solo nodo primario**. Questa struttura basata su un singolo nodo abilitato a gestire tutte le operazioni di scrittura, assicura una consistenza rigorosa (**strict consistency**) dei dati presenti nelle varie repliche.

Per quanto concerne le operazioni di lettura, queste possono essere indirizzate a qualsiasi nodo, sebbene, per default, siano indirizzate solo a quello primario. Ciò garantisce che le operazioni di lettura restituiscano sempre la **versione più aggiornata disponibile** di un data (**documento**) al costo però di una **riduzione** (anche significativa) del **throughtput** e, in alcune contesti, un **aumento della latency** (per esempio un client situato a Londra deve accedere al nodo primario ubicato in un data center di New York). In questa configurazione, i nodi secondari sono utilizzati principalmente per estendere lo spazio dati a disposizione e per implementare feature come elevata disponibilità e fault tolerance.

L'abilitazione delle **letture da nodi non primari** migliora notevolmente la scalabilità, (potenzialmente) il throughput del sistema e, in alcuni contesti, riduce la latency delle operazioni in

lettura, a spese però del **rischio** che le applicazioni client leggano dati dai vari nodi non ancora aggiornati. In questo contesto di parla di sistemi “eventualmente” consistenti. La configurazione migliore, come al solito dipende dai requisiti o meglio da un corretto bilanciamento tra requisiti e soluzioni architetturali.

Configurazione delle letture

Per quanto concerne la configurazione delle letture, MongoDB può essere impostato con le seguenti cinque modalità di lettura (**read preference**):

1. primary
2. primaryPreferred
3. secondary
4. secondaryPreferred
5. Nearest

Questa modalità può essere impostata direttamente negli **oggetti** di **connessione**, in quelli di tipo **database**, in quelli di **collezione**, oppure direttamente nelle **singole operazioni**. Vediamo brevemente di descrivere le cinque modalità di lettura di un cluster MongoDB.

primary è la modalità di default. Oltre alle operazioni di scrittura, anche le operazioni di lettura sono indirizzare esclusivamente al nodo primario. Qualora questo non sia disponibile, l’operazione genera un’apposita eccezione.

primaryPreferred fa sì che, nella maggior parte dei casi, le operazioni di lettura siano inoltrate al nodo primario. Tuttavia, in situazioni particolari come la non disponibilità del nodo primario, le

operazioni di lettura sono indirizzate ai nodi secondari.

secondary, come si intuisce dal nome, implica che tutte le operazioni di lettura siano inoltrate ai nodi secondari. Qualora nessu secondario sia disponibile, l'operazione genera un'eccezione.

secondaryPreferred indirizza le operazioni di lettura ai nodi secondari; tuttavia, qualora questi non siano disponibili, le letture sono inoltrate al nodo primario.

Nearest funziona in base alla distanza e non al fatto che il nodo sia primario o secondario: le operazioni di lettura sono indirizzate al nodo più vicino, indipendentemente dalla sua tipologia (primario o secondario).

A conclusione di questa spiegazione, va detto che le applicazioni **client**, e i **nodi** membri del gruppo di **repliche** di MongoDB, eseguono periodicamente e in modo trasparente (attraverso il driver), un **processo** utilizzato che si occupa di **determinare/mantenere aggiornato** lo stato del cluster e che consiste in liste dei vari nodi. Grazie alla disponibilità di tali **liste**, i nodi client sono in grado di individuare il nodo più vicino nell'elenco dei nodi secondari, e quello primario. Queste informazioni sono il prerequisito per poter mettere in atto le varie strategie di lettura: senza tali informazioni, infatti, non sarebbe possibile stabilire qual è il primario, quali sono i secondari, e quale il nodo più vicino.

Operations log: oplog

La **sincronizzazione** tra il nodo **primario** e le varie **repliche** avviene per mezzo di uno speciale file di log, denominato **oplog** ("operations log", cfr. figura 1). In particolare, il nodo primario vi

annota tutte le operazioni di aggiornamento della propria base dati e i membri secondari hanno il compito di copiare e applicare asincronicamente le operazioni presenti nel log. Ciò è possibile giacche' tutti i membri del gruppo delle repliche ricevono una copia dell'**oplog** che gli permette di mantenere allineato lo stato del database. Le operazioni sono specificate nel log in modo da essere **idempotenti** (producono il medesimo risultato sia se eseguite una sola volta, sia un numero variabile). Ciò fa sì che lo stesso **log** possa essere eseguito un numero variabile di volte da un medesimo nodo portando la base dati al medesimo stato.

Organizzazione dei dati in MongoDB

MongoDB organizza i dati secondo una **struttura gerarchica** (figura 2). In particolare, ogni istanza può avere diversi database; ognuno di questi, tipicamente, possiede diverse collezioni i cui membri sono i documenti. Questi ultimi contengono diversi campi.

Figura 2 – Organizzazione gerarchica dei dati.

Volendo fare un **paragone** con i **database relazionali**, è possibile associare i seguenti concetti tra loro: un database tradizionale equivale ad un oggetto database di MongoDB. Le tabelle e le viste invece possono essere associate al concetto di Collezione. Come visto in precedenza, in MongoDB non ci sono righe (row) e i dati sono collocati in appositi Document memorizzati attraverso la

codifica BSON. Le colonne dei database tradizionali corrispondono ai campi presenti nei document. Entrambi i sistemi prevedono la presenza degli indici. Una partizione di un database tradizionale è rappresentata da uno Shard (questo concetto viene discusso nei paragrafi seguenti). Le tradizionali Join sono rappresentate in MongoDB come documenti annidati.

La tabella 1 riassume in forma schematica i concetti appena esposti.

Tabella 1 – Equivalenza tra i concetti dei tradizionali database relazionali e MongoDB.

Partizionamento (sharding)

Lo **sharding** è il metodo adottato dai sistemi distribuiti per **dislocare dati** su diversi **nodi/server**. Si tratta di una delle strategie utilizzate per permettere ai vari sistemi di gestire vaste e vastissime quantità di dati fornendo, al tempo stesso, un elevato throughput alle operazioni sui dati.

Lo **sharding** appartiene alle tecniche di **scalabilità** definite **orizzontali**, in “contrapposizione” a quelle **verticali** che si basano essenzialmente sull’aggiunta di **risorse** come la memoria, il numero di CPU e di server, e così via. In sistemi reali, come è lecito attendersi, elevate scalabilità possono essere ottenute solo combinando le due tecniche.

La tecnica dello **sharding**, che in MongoDB opera al livello di **collezione**, si occupa di partizionaree un intero insieme di dati e di distribuire le varie partizioni sui nodi disponibili (i membri del gruppo di repliche). In questo modo, ogni “collezione partizionata” (**shard**, non necessariamente ogni nodo) può essere considerata come un “database” indipendente e la **somma** dei vari **shard** forma il singolo **database globale**. Chiaramente, la partizione non è mai perfetta come mostrata in figura 3: i vari nodi devono necessariamente contenere anche dati appartenenti a shard diversi al fine di fornire la necessaria ridondanza.

Figura 3 – Strategia di sharding da un punto di vista logico.

In MongoDB generalmente un nodo gestisce più shard; il corretto funzionamento di un gruppo di shard necessita dei seguenti componenti: **Shard**, **Query router** e **Config server** (figura 4).

Figura 4 – Collaborazione tra componenti Query router e Config server.

Uno **shard** è una **partizione** di una **collezione**, e quindi si occupa

di memorizzare la propria porzione di dati. I vari shard sono distribuiti nei gruppi di repliche.

Query router e Config server

I **Query router** ("instradatori di interrogazione") sono processi di tipo mongod, che si occupano di ricevere le richieste da parte dei client di MongoDB e di ridirigerle allo shard, o al sottoinsieme di shard in grado di soddisfare la richiesta. Chiaramente si occupano di consolidare le varie risposte da inviare indietro al client. Tipicamente, per ogni shard esiste un insieme di Query Router che operano su di esso.

I **Config server** ("sever di configurazione") si occupano di memorizzare metadati relativi ai cluster, come per esempio il mapping dell'allocazione dei dati ai vari shard. Si tratta di informazioni vitali per il Query router per poter ottemperare ai propri compiti.

Algoritmo di partizionamento e shard key

La partizione dei dati avviene attraverso l'applicazione di un determinato **algoritmo** a un campo chiave, che in MongoDB è chiamato **shard key**. Questo campo può essere una chiave primaria sia singola sia composta, l'importante è che i campi che la compongano siano presenti in tutti i documenti che appartengono alla particolare collezione. Di fatto, i campi costituenti la **shard key** sono obbligatori.

Il ruolo dell'**algoritmo** consiste nel generare delle **partizioni bilanciate** della **collezione** chiamate **chunks** ("pezzo", "parte"). A tal fine MongoDB può utilizzare due strategie: l'**hashing** o il

range.

Per quanto concerne la prima, si tratta di una soluzione classica del tutto equivalente a quella descritta per Oracle Coherence. In sostanza, attraverso l'applicazione della funzione di hashing, ogni key viene fatta corrispondere a un campo numerico che rappresenta l'indice logico dello shard di appartenenza. Questo indirizzo viene ulteriormente manipolato per normalizzarlo ed alla fine si applica un'operazione di modulo per individuare lo specifico shard.

Gli algoritmi **range-based** (basati su un campo di valori) sono molto simili alla strategia comunemente utilizzata negli elenchi alfabetici: l'intero spazio dei valori è organizzato in gruppi/chunk (l'iniziale dell'elenco in ordine alfabetico A, B, C, …) e ogni chiave appartiene a un gruppo specifico.

Al fine di mantenere una distribuzione omogenea dei dati, MongoDB utilizza due processi eseguiti in background denominati **Splitting** ("suddivisione") e **Balancing** ("bilanciamento").

Il primo, tipicamente azionato da operazioni di inserimento e aggiornamento, si occupa di controllare che i vari raggruppamenti (**chunk**) non crescano oltre un determinato valore di soglia (**chunk size**). Qualora questo valore sia raggiunto, l'algoritmo procede con la suddivisione del chunk in sottogruppi di uguali dimensioni. Tale processo tuttavia non esegue la migrazione dei dati tra shard.

Questo è invece compito del processo di **bilanciamento**. Quando la distribuzione di una collezione presente in uno shard diviene irregolare, il processo di bilanciamento si occupa di far migrare i blocchi dal frammento che ha il maggior numero di elementi verso

il frammento con il minor numero di blocchi finche' non venga rispristinata una **situazione bilanciata**. Per esempio: se una collezione ha 100 membri allocati in un primo shard (shard 1) e 50 su un'altro (shard 2), il servizio di bilanciamento si occupa di migrare circa 25 membri dallo shard 1 allo shard 2 generando un nuovo equilibrio sulla distribuzione della collezione. Come è lecito attendersi questo processo viene eseguito in background.

Processo di elezione del nodo primario

L'architettura di MongoDB è fortemente centrata sul **nodo primario**: qualora questo divenga non più disponibile, scenario molto serio, l'insieme di repliche esegue nuovamente il processo di **elezione** del nodo primario (**replica set election**). Questo processo è eseguito non solo nel contesto del processo automatico di failover, ma anche alla costituzione iniziale del gruppo di repliche e quando si aggiunge uno o più nodi a priorità più elevata dell'attuale primario (le priorità sono spiegate di seguito). Durante l'esecuzione di questo processo, non sempre rapido (in un contesto reale, la scoperta della non disponibilità del nodi primario può richiedere 30 secondi, mentre per il recovery si parla di minuti), il gruppo di repliche diviene **non disponibile**.

La strategia utilizzata per scoprire che un nodo non è più disponibile è lo **heartbeat**. In MongoDB i vari nodi si scambiano un "battito cardiaco" ogni due secondi. Qualora un membro non risponda entro dieci secondi, i nodi lo marcano come inaccessibile. Se poi si tratta proprio del nodo primario, allora si procede alle elezioni basate sulla **priorità**, attributo opzionale presente nel documento di configurazione dei nodi del cluster. Si tratta di un numero che può assumere valori compresi tra **0** e **100.0**, con priorità crescente: più il valore è alto e maggiori sono le

possibilità per il nodo di diventare primario. Un valore pari a zero indica sia l'impossibilità di un nodo di diventare primario, sia di avviare il processo di elezione anche nella condizione in cui il nodo non riesca ad instaurare una connessione con il nodo primario.

Impostare la priorità di alcuni nodi a zero permette di dar luogo ad una configurazione particolarmente utile atta a **marcare alcuni nodi**, magari ubicati in data center secondari, **non** idonei a diventare **primari**. Questi nodi, inoltre, non sono abilitati ad avviare un processo di elezione qualora rilevino lo scenario di impossibilità di collegamento con il nodo primario. Questo è utile per evitare che avviino la procedura di elezioni solo perche', per esempio, la connessione tra i vari centri presenta dei ritardi. In ogni modo, questi nodi hanno lo scopo principale di mantenere copie dei dati, tipicamente al fine di accettare operazioni di lettura da clienti locali e hanno il diritto di partecipare alle elezioni.

Ogni membro dell'insieme di repliche cluster ha una **priorità** che consente di determinare la sua idoneità a diventare un primario. In un'elezione, il gruppo di repliche elegge un membro come primario tra quelli ammissibili con il valore più alto di priorità. Un membro del set di repliche può diventare primario (appartiene al gruppo di quelli ammissibili) solo se può **connettersi** con la maggioranza dei membri del set di repliche.

Tutti i membri hanno una priorità; per default è impostata pari ad uno. È possibile impostare il valore di priorità per appesantire le elezioni a favore di un particolare membro o di un gruppo dell'insieme di repliche. Per esempio, se si dispone di un set di repliche geograficamente distribuito, è possibile **regolare** le **priorità** in modo tale che solo i **membri** di un **determinato data center** possano diventare **primario**. Il primo membro a ricevere la

maggioranza dei voti diventa primario. Per default, tutti i membri hanno un solo voto, a meno che non si modifichi tale impostazione. È infatti possibile impostare tale selezione in modo tale che ci siano nodi che non abbiamo diritto al voto (**votes=0**). Da notare che si tratta di un concetto diverso della priorità pari a zero.

I vari compromessi possibili

Il teorema di Brewer afferma che non è possibile ottenere un database distribuito che allo stesso tempo sia C ("consistente"), A ("ad alta disponibilità") e P ("tollerante alle partizioni"). Vediamo di seguito le diverse strategie possibili con MongoDB.

Compromesso CP

Come visto nei paragrafi precedenti, quando si utilizzano le impostazioni di default (**ReadPreference = primary**), MongoDB è fortemente **consistente**. Infatti, rispetta le due regole base:

- tutti i nodi replica contengono la stessa versione dei dati (ciò ovviamente non significa che contengano tutti i dati);
- i clienti del sistema ottengono la stessa versione dei dati indipendentemente dal nodo a cui inoltrano (o tentano di inoltrare) la richiesta.

In termini molto pratici ciò significa che, se si esegue un'operazione di scrittura di un determinato dato e immediatamente dopo si esegue si esegue una lettura dello stesso, assumendo ovviamente che la scrittura abbia avuto esito positivo, si è sempre in grado di leggere il risultato della scrittura appena effettuata. Ciò perche' l'architettura di MongoDB è basata

su un'organizzazione che, in ogni momento, prevede un singolo nodo che viene eletto come **primario** e tutte le operazioni, letture incluse, sono gestite dal nodo primario.

Inoltre, MongoDB è tollerante alle **partizioni** in quanto il sistema rimane operativo nel caso in cui avvenga una partizione del network. Tuttavia, l'implemetazione di una struttura master/slave fa sì che via sia una volubilità localizzata nel nodo primario.

Compromesso AP o solo P

Oltre al funzionamento di default, MongoDB prevede un modello di funzionamento che permette ai clienti di eseguire letture direttamente dai nodi secondari. In questo caso, si **perde** la feature di **consistenza** rigorosa. In effetti, nell'arco di tempo necessario tra il momento in cui un dato è inserito nel sistema nel nodo primario e quello in cui è applicato dai vari nodi secondari vi è una latenza. Durante tale latenza, le letture dello stesso dato su nodi secondari restituscono un dato non aggiornato, stale. Quindi non si può più parlare di consistenza rigorosa. Tuttavia, in questo contesto, parlare di compresso **AP** non sembra proprio corretto. In effetti, l'elevata **availability**, sempre dipendente dal nodo primario, non migliora e quindi, molto probabilmente, questo modo di operare porta più ad un compresso di tipo solo **P**.

Conclusione

Abbiamo iniziato questa serie presentando il teorema di Brewer, detto anche di CAP che brevemente sancisce che **non è possibile** per un sistema software distribuito presentare **simultaneamente** le tre seguenti garanzie: **consistenza**, **continua disponibilità** e **tolleranza alle partizioni**.

Chiaramente, questa limitazione è apprezzabili solo in sistemi soggetti a requisiti non funzionali non ordinari.

Questo teorema, considerato una pietra miliare anche del pensiero NoSQL, dovrebbe essere assolutamente chiaro a tutti gli architetti. Non solo il teorema stesso, ma anche e soprattutto le sue ripercussioni pratiche sul disegno di sistemi distribuiti. Ogni architetto deve essere in grado di comprendere, fin dalle primissime fasi di un progetto, se i requisiti non funzionali richiedano soluzioni appartenenti al dominio dei sistemi non fattibili. Non c'è cosa peggiore di lavorare a un progetto, investire ingenti risorse in termini di tempo e di bugdet, per poi rendersi conto di trovarsi in un contesto di soluzioni non fattibili o addirittura di stare tentando di implementare una ennesima soluzione NoSQL.

Come **esempi pratici** delle scelte architetturali dettate dal teorema abbiamo descritto due esempi: **Oracle Coherence** e **MongoDB**. Oracle Coherence è un data-grid internamente gestito in memoria, sempre disponibile grazie alla possibilità di ricorrere a un elevato cluster di computer, adatto per l'analisi dei dati in tempo reale, calcoli di tipo grid eseguiti in memoria, e soluzioni ad alte prestazioni. Si tratta di una soluzione ascrivibile al gruppo delle soluzioni **AC**. Come visto, in Coherence lo scenario pericoloso è dato dalla situazione di partizionamento del cluster (split-brain): i nodi si organizzano in sotto cluster autonomi, credendo che i nodi degli altri, non più raggiungibili, siano morti. Questa configurazione è pericolosa in quanto può portare alla perdita di dati.

MongoDB è un database **NoSQL**, **document-oriented** a elevate performance, ad alta disponibilità in grado di scalare semplicemente. Un documento può contenere numerose coppie

chiave-valore (key, value pairs), o coppie chiave-array o anche documenti annidati. In MongoDB, le strutture ed i dati sono rappresentate per mezzo di una codifica standard, denominata BSON (Binary JSON) in quanto ottimizzazione della codifica JSON.

La scelta operata dal team di MongoDB è stata di orientarsi su un compromesso di tipo **CP**. Quindi può offrire un'elevata coerenza dei dati, ed è in grado di tollerare le partizioni. Tuttavia, data l'architettura master/slave **non** presenta un'elevata **disponibilità** (availability). In contesti reali, il cluster può impiegare fino a 30 secondi per rendersi conto che il nodo primario non è più disponibile, e diversi minuti per recuperare automaticamente il corretto funzionamento.

Oltre le specifiche soluzioni, è interessante notare le soluzioni e gli algoritmi implementati dai vari team per la distribuzione dei dati (per esempio hashing con e senza configurazione master/slave), per la rilevazione del corretto funzionamento dei cluster (tutte e due le soluzioni utilizzano la tecnica del "battito cardiaco"), gli algoritmi di voto/elezione dei nodi primari, le procedure di recovery, etc. Si tratta di strategie e disegni che devono essere comprese per utilizzare questi tool e che poi fanno parte del patrimonio di conoscenze degli architetti.

Con l'avanzata delle soluzioni NoSQL, l'egemonia dei database relazioni non è più così incontrastata. Una notevole accelerazione si è avuta con l'evoluzione del Web2.0. In particolare, importanti servizi Internet si sono trovati a far fronte ad una crescita esponenziale dei dati da manipolare. Crescita dovuta essenzialmente all'espansione di Internet sia in termini della crescita dei siti web sia in termini della popolazione utenti. Tale espansione vertiginosa ha finito per generare volumi di

transazioni tali da evidenziare tutti i limiti delle soluzioni tradizionali basate su i database relazionali.

Intorno al 2008-9, Google con Bigtable e Amazon con Dynamo si sono trovati a far da apripista a questo nuovo movimento NoSQL. A questo punto ci si potrebbe domandare se il dominio dei database relazionali è veramente sotto attacco? Attualmente si tratta ovviamente di una domanda retorica. Tuttavia nelle organizzazioni è sempre più frequente trovare database NoSQL che affiancano i tradizionali database relazionali in aree caratterizzate da pressanti requisiti non funzionali. Ad oggi i database NoSQL sembrano essere un completamento dell'offerta di soluzioni per la manipolazione dati, piuttosto che un rimpiazzo dei database relazionali.

Riferimenti

[1] Luca Vetti Tagliati, "Il teorema CAP… in Brewer – IV parte: I database NoSQL", MokaByte 189, novembre 2013

https://www.mokabyte.it/cms/article.run?articleId=X7V-49R-NDX-ZCG_7f000001_11231952_6c4926fa

[2] Luca Vetti Tagliati, "Il teorema CAP… in Brewer – I parte: Il teorema CAP (Consistency, Availability, Partition tolerance)", MokaByte 186, Luglio/Agosto 2013

https://www.mokabyte.it/cms/article.run?articleId=4FI-FRM-HJB-RTZ_7f000001_11885319_d2be079e

[3] MongoDB Documentation, Release 2.4.8, MongoDB Documentation Project, novembre 2013

http://docs.mongodb.org/manual/