mokabyte.it

Il teorema CAP... in Brewer - MokaByte

Luca Vetti Tagliati

31-41 minuti

Continuiamo la serie dedicata al teorema CAP, o di Brewer, approfondendo la cache distribuita Oracle Coherence. In particolare, dopo aver ripassato il teorema nel corso dell'articolo precedente, in questo presentiamo Oracle Coherence nelle sue caratteristiche. Nel corso del prossimo articolo illustreremo dettagliatamente le soluzioni architetturali e le scelte di disegno originate proprio dal teorema CAP.

Introduzione

Nell'articolo precedente abbiamo visto il teorema **CAP** (**Consistency**, **Availability**, **Partition tolerance** ossia totale consistenza, continua disponibilità e tolleranza alle partizioni), detto anche teorema di Brewer, dal nome della persona che per primo lo ha congetturato [1]. Il teorema afferma che è impossibile per un sistema informatico distribuito fornire **simultaneamente** tutte e tre le seguenti garanzie: completa **consistenza** dei dati, continua **disponibilità** e **tolleranza** alle partizioni.

In questo articolo presentiamo gli aspetti fondamentali di Oracle Coherence: "a continuously available in-memory data grid across a large cluster of computers for real-time data analysis, in-memory grid computations and high-performance processing solution" ("data-grid internamente gestito in memoria, sempre disponibile grazie 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"). L'obiettivo è fornire i concetti fondamentali per poter evidenziare, nel corso del

prossimo articolo, l'influenza del teorema CAP sulle scelte architetturali e di disegno di Oracle Coherence.

In particolare, dopo aver presentato brevemente la storia di questo **data grid**, focalizziamo l'attenzione sulle caratteristiche principali di Oracle Coherence senza scendere eccessivamente nei dettagli, cosa che invece avviene nel corso del prossimo articolo dove evidenziamo i dettagli della scelta di disegno di Coherence di implementare il **compromesso** di tipo **AC** (scelta assolutamente compatibile con il relativo DBMS). Questo implica capacità di fornire la **coerenza** dei dati, da cui deriva peraltro il nome Coherence, e l'elevata, quasi continua, **disponibilità** a discapito però della tolleranza alla partizioni che ne rappresenta il punto debole. Lo scenario più pericoloso illustrato nel prossimo articolo, è il cosiddetto split-brain ("cervello diviso"), ossia la creazione di due sotto-cluster autonomi, che, in situazioni limite, può causare la perdita di dati.

Breve storia di Coherence

Le prime versioni di Coherence, **cache distribuita** a **elevate prestazioni**, furono implementate da un'azienda relativamente piccola chiamata Tangosol, la quale, grazie proprio a Cohernce divenne, intorno al 2002-2004, uno dei leader nel mercato di fornitori cache distribuite.

Il 23 marzo del 2007, **Oracle** annunciò l'acquisizione della società Tangosol [2] al fine di incrementare la propria offerta della famiglia prodotti Fusion Middleware. Questa acquisizione rientrava nel contesto della strategia di Oracle di perfezionare il proprio portfolio di prodotti e infrastrutture software necessarie per penetrare ulteriormente in settori chiave come quelli finanziari/bancari, e quelli per telecomunicazioni, viaggi, scommesse e giochi online.

Una considerazione importante di questa acquisizione, spesso sottovalutata, è di vedere un leader di soluzione **RDBMS** (Relational Database Management System), probabilmente il principale, che giunge a una conclusione: la tecnologia di **database tradizionale non** era più

una piattaforma **sufficiente** per rispondere alle esigenze specifiche di una vasta classe di requisiti e di architetture, come ad esempio eXtreme Transaction Processing (XTP) e analisi in tempo reale di grandi quantità di data [3].

L'acquisizione di Coherence è stata una mossa strategica volta ad affrontare diverse criticità Oracle, tra le quali le più importanti sono **prestazioni**, **scalabilità**, **semplicità**.

Performance

Soluzioni basate esclusivamente su RDBMS, in situazioni di **latenze bassissime**, possono presentare **performance non ottimali**. Ciò è dovuto al fatto che il database mantiene lo stato transazionale e tutti gli altri dati su un disco fisico, pertanto i **dati** sono molto **distanti** dalla logica applicativa: è una situazione problematica per i sistemi a bassissima latenza. Inoltre i dati sono molto distanti anche da altre componenti tipicamente presenti nella maggior parte dei sistemi software, come ad esempio la messaggistica. Soluzioni RDBMS tradizionali che cercano di portare la logica di business più vicino ai dati, ad esempio **stored procedures**, hanno finito per creare più problemi che vere soluzioni.

Scalabilità

Applicazioni interamente centrate sul database tendono, con l'**aumento** del **carico** di lavoro, a presentare problemi di **scalabilità**. Si tratta di un fattore intrinseco nell'architettura: tutte le richieste finiscono per richiedere l'intervento della banca dati che inevitabilmente diventa il collo di bottiglia. Anche se tutta una serie di nuove soluzioni sono state introdotte nello spazio RDBMS, questo resta ancora fondamentalmente un **servizio centralizzato**, non sempre adatto per la gestione di grandi volumi di dati maneggiati dai sistemi attuali di grandi organizzazioni come le banche.

Semplicità

I RDBMS **non** sono intrinsecamente la soluzione **più compatibile** per costruire sistemi Object Oriented, anche se **SOA** prima e **Big Data** attualmente stanno minando alle basi questo paradigma. Sebbene esistano molteplici framework e architetture disegnate per affrontare la **mappatura Relazionale-OO**, questa soluzione non è ancora così semplice come trattare direttamente con **oggetti** presenti in memoria. Inoltre, poiche' la logica business è gestita da uno strato del sistema e i dati sono mantenuti in un'infrastruttura diversa, il **clustering** (includendo il partizionamento, il bilanciamento del carico di lavoro e fail-over) per **ogni strato** deve essere gestito separatamente, aggiungendo complessità per gli sviluppatori. Detto questo, è importante sottolineare che anche l'introduzione di implementazioni non banali di cache distribuite tende ad aggiungere nuove aree di complessità, e può portare a sistemi più impegnativi da supportare in produzione.

Che cosa è Oracle Coherence?

Come riportato sopra, 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. Ma vediamo cosa si cela dietro questa ampollosa definizione.

Grid dati

Una **grid** è un **insieme di server** che **cooperano** al fornire un insieme di funzionalità ben definite. In questo caso, si tratta di un **grid dati**, pertanto i server cooperano al fine di gestire un insieme di informazioni; tradotto in termini Object Oriented, significa una serie di **grafi di oggetti**, quelli utilizzati dalle varie applicazioni: per questo spesso si parla di **application objects**.

Serializzazione

Coherence mantiene questi oggetti in memoria, nello spazio **heap** delle varie JVM che partecipano al cluster, opportunamente **serializzati**. Sebbene sia possibile scegliere tra diverse soluzioni di serializzazione, Oracle raccomanda di utilizzare il formato **POF** (Portable Object Format, "formato oggetto portatile" [4]). Si tratta di un formato **binario**, indipendente dal linguaggio di programmazione, disegnato per essere efficiente e molto leggero, al fine di ridurre l'occupazione di memoria ("memory footprint") e l'impatto delle operazioni dei processi di serializzazione/deserializzazione sulle performance.

Memoria

Poiche' la **memoria** è una risorsa volatile di per se', ed è assolutamente vulnerabile, Coherence adotta una ben collaudata strategia di **ridondanza** per fornire sia un'elevata disponibilità dei dati, sia la necessaria resilienza a possibili fallimenti hardware (e non). Tale strategia si basa sulla distribuzione di **copie** degli **stessi grafi** di oggetti su **più server**, mantenute opportunamente **sincronizzate**.

Clustering

Il protocollo di **clustering** utilizzato è in grado di fornire i dati ai vari server con performance molto elevate, comparabili alla latenza tipica della sola trasmissione via cavo. La ridondanza dei dati permette a Coherence di fornire i necessari requisiti di **failover** e **failback**, senza alcuna perdita di dati. Il servizio di cache distribuita permette di configurare il numero di backup: chiaramente se tale numero è impostato a uno, il singolo nodo del cluster può fallire generando perdita di dati. Coherence distribuisce gli oggetti tra i vari nodi del grid utilizzando un algoritmo tipicamente basato sullo **hashing** della chiave dell'oggetto: i grafi di oggetti possono essere aggiunti nelle proprie cache attraverso un'interfaccia tipo **java.util.Map**.

L'**aggiunta** di nuovi **nodi** al cluster fa sì che vi sia una **riorganizzazione** della **distribuzione** dei dati mantenuti in cache al fine di ripartire

opportunamente l'insieme degli oggetti tra i vari nodi: ognuno si assume la responsabilità per la sua parte dell'insieme totale. Si ottiene quindi il **bilanciamento** del **carico** di **lavoro** tra i vari nodi, ognuno dei quali si trova a gestire porzioni di dati sempre minori. Questa soluzione permette a Coherence di rimuovere/minimizzare colli di bottiglia e di ottenere latenze molto basse e predicibili (lineari come descritto di seguito) con l'aumento del numero di server.

Anche se l'informazione è distribuita su più server, Coherence garantisce la fondamentale caratteristica di **trasparenza dalla locazione**: le applicazioni utilizzano la stessa identica API indipendentemente dalla loro ubicazione "fisica" ossia dal nodo/server che ha l'oggetto nella propria area primaria. Gli sviluppatori, quindi, non devono farsi carico di scrivere del codice per individuare dove l'oggetto desiderato è memorizzato, ne' hanno necessità di essere a conoscenza degli aspetti topologici della cache. L'API e il suo comportamento sono uguali sia se si ha a che fare con una cache totalmente locale sia con un sistema interamente replicato (cache distribuita). Chiaramente quello che cambia sono le performance.

Mappe di oggetti distribuiti

Dal punto di vista implementativo, Coherence può essere considerato, in termini intuitivi, come un insieme di **mappe di oggetti distribuiti** in grado di memorizzare **coppie chiave-valore**, dove il valore dell'elemento è, nella stragrande maggioranza dei casi, un ben definito grafo di oggetti opportunamente serializzato. In Coherence, ogni **mappa** è un oggetto chiamato **NamedCache** da recuperare tramite un nome univoco, per esempio:

```
NamedCache map = CacheFactory.getCache ("FinancialInstruments");
```

La classe **NamedCache** implementa l'interfaccia **java.util.Map** ottenendo un altro importante vantaggio: gli sviluppatori non devono imparare una nuova API. Nel libro [5] vi è un intero capitolo dedicato all'argomento hashing.

La trappola dell'implementazione in-house

Da un punto di vista molto semplificato, Coherence può essere pensato come un **database in memoria**, organizzato in una serie di **tabelle identificate** da un **nome**, in cui ciascuna tabella è una **mappa**. Questa definizione è utile da un punto di vista intuitivo ma non deve assolutamente generare nei lettori l'impressione che implementare una cache sia un esercizio immediato e che quindi sia una buona idea implementare soluzioni in-house di cache tramite **java.util.HashMap**. Questa è una trappola pericolosa dove anche programmatori esperti a volte finiscono per cadere.

È importante ricordare che soluzioni di cache idonee per poter essere inserite in sistemi in produzione richiedono tutta una serie di servizi complessi quali integrazione tra i nodi della rete a **bassissima latenza**, **partizionamento dinamico**, sofisticate politiche di **sfratto dei dati**, possibilmente eseguito in background, **lock** dei vari **oggetti**, avanzate strategie di **overflow su disco**, supporto per diversi modus operandi, ad esempio write-behind, possibilità di reperire gli oggetti per mezzo di linguaggi **query di alto livello**, servizi di **amministrazione**, etc. Per questo motivo l'esperienza insegna che tentativi di implementare soluzione **cache in-house** finiscono per codificare solo la punta dell'iceberg. Queste "cache" semplicistiche implementate tramite **java.util.HashMap** altro non sono che un semplici blog in memoria. Data l'ampia offerta di soluzioni di cache che include soluzioni open source e soluzioni di vendor, tutti dovrebbero pensarci mille volte prima di intraprendere la realizzazione di una cache ad hoc anche per scopi semplici.

Come funziona

Coherence gestisce i dati attraverso un insieme di mappe (**NamedCache**) partizionate su diversi nodi del cluster, tipicamente in funzione su vari server, secondo un'opportuna strategia. Ciò significa che gli oggetti gestiti da ciascuna mappa sono **distribuiti** tra i vari nodi in

modo che esattamente solo uno di loro sia responsabile (**primario**) di ogni oggetto. L'allocazione degli oggetti ai nodi del cluster è ottenuto attraverso un bilanciamento implicito del carico basato sullo **hashing** della chiave dell'oggetto (l'oggetto usato come key durante l'aggiunta alla mappa) e sulla sua **distribuzione** normale.

Figura 1 – Vista concettuale della mappa e allocazione fisica degli oggetti ai nodi.

Struttura logica di un nodo

In particolare, per ogni oggetto Coherence definisce un **nodo proprietario primario** e uno o più **nodi di backup**. Il diagramma di figura 2 mostra la struttura logica di un nodo del cluster. Sebbene da un punto di vista concettuale, grazie alla proprietà di trasparenza dalla posizione, ogni nodo contiene tutti gli oggetti gestiti dalla cache, in realtà ogni nodo è il proprietario principale di un sottoinsieme di oggetti assegnato dinamicamente (oggetto A nel diagramma) ed è il nodo di backup per un diverso sottoinsieme di dati (oggetto B nel diagramma).

Figura 2. Struttura logica di un nodo del cluster.

Operazione di put

Consideriamo il processo di aggiunta di un nuovo oggetto nella cache, l'operazione di "put". Il listato seguente mostra un semplice esempio in cui un nuovo strumento finanziario è aggiunto al corrispondente oggetto **NamedCache**.

```
NamedCache financialInstrumentCache =
   CacheFactory.getCache("FinancialInstrument");
 financialInstrumentCache.put(
   aFinancialInstrumentVO.getInstrumentKey(),
   aFinancialInstrumentVO);
```

Nella figura 3, l'operazione di aggiunta di un nuovo oggetto nella cache viene rappresentata in modo schematico.

Figura 3 – Operazione di put: aggiunta di un nuovo oggetto nella cache.

Supponiamo che l'applicazione debba aggiungere l'**oggetto D** nella cache (figura 3) e pertanto esegua un'operazione di **put** dell'oggetto **D**. In questo caso, esso viene memorizzato nella partizione primaria dello stesso nodo (**JVM – Nodo 1**). Tuttavia questo non è sempre il caso, anzi: infatti, come specificato in precedenza, il nodo primario dipende dall'hash code della sua chiave (**key.hashCode()**). Assumendo che resti nello stesso nodo, **JVM – Node 1** diviene il proprietario principale di **D**.

Al fine di assicurare **affidabilità** e **failover**, lo stesso oggetto, opportunamente serializzato, è inviato in modo trasparente a un altro nodo del cluster (**JVM – Nodo 2**) che quindi lo memorizza nella propria zona di backup. In questo scenario, l'**oggetto D** è inviato ad un solo nodo di backup (**contatore** di backup impostato al valore di default pari a **1**). Tuttavia, lo scenario decimante più frequente prevede che la cache distribuita sia adottata in contesti dove requisiti non funzionali quali alta

disponibilità e resilienza sono critici.

In questo scenario, il contatore assume un valore superiore che fa sì che la copia dell'oggetto sia inviata a diversi nodi per memorizzarlo nelle proprie aree di backup. Il numero dei nodi di backup, come è lecito attendersi, ha un impatto sulle prestazioni delle operazioni di modifica della cache: queste devono essere replicate nei vari nodi backup. Inoltre, configurazioni classiche prevedono che tali operazioni di modifica siano considerate complete solo quanto tutti i nodi di backup hanno confermato la ricezione della modifica. Pertanto, vi è un prezzo in termini di prestazioni (a dire il vero molto basso grazie al protocollo di rete ad alte prestazioni utilizzato) da pagare per le modifiche della cache in presenza di diversi nodi di backup.

Ciò fornisce la garanzia fondamentale di mantenere la consistenza dei dati nel caso in cui un nodo del cluster fallisca inaspettatamente o risulti irraggiungibile per problemi di rete. In tal caso, i dati di backup di determinati nodi sono promossi nello storage primario. È la cache distribuita (attraverso un opportuno protocollo descritto di seguito) che come al solito si occupa di coordinare il lavoro di tutti i nodi del cluster rimanenti per riallocare la responsabilità primaria degli oggetti che erano responsabilità del nodo fallito, residenti nelle zone di backup dei vari nodi.

Operazione di get

Una volta che i dati sono stati memorizzati nella cache, è possibile recuperarli utilizzando due API differenti: reperimento attraverso la **chiave primaria** secondo la classica interfaccia **Map** e utilizzando l'apposito **linguaggio di query**. Iniziamo con il considerare il primo caso.

```
NamedCache financialInstrumentCache =
CacheFactory.getCache("FinancialInstrument");
  aFinancialInstrumentVO =

(FinancialInstrumentVO)financialInstrumentCache.get(financialInstrumentKey);
```

Figura 4 – Operazione di get: ottenimento di un nuovo oggetto dalla cache.

Si supponga che l'applicazione in esecuzione su **JVM – Node 1** debba recuperare l'**oggetto A** (figura 4) che, nel caso specifico, è memorizzato nello stesso nodo (stessa JVM). In questo caso, l'operazione è estremamente semplice, non comporta chiamate di rete e l'oggetto viene restituito immediatamente, dopo essere stato opportunamente deserializzato.

La situazione è leggermente diversa quando l'applicazione ha bisogno di recuperare l'**oggetto C**. In questo caso, l'oggetto è gestito da un nodo della cache in esecuzione su una diversa JVM. Poiche' ogni oggetto è responsabilità (primaria) di un solo nodo del cluster, questo scenario richiede un accesso in rete: una singola operazione di andata e ritorno (**2 hops**). Queste comunicazioni in Coherence sono tipicamente estremamente scalabili ed efficienti poiche' viene utilizzata una comunicazione diretta di tipo **point-to-point**.

Coherence richiede l'**oggetto C** al nodo in esecuzione su **JVM – Node 2** e lo restituisce all'applicazione. Tutto avviene in modo trasparente, l'unica differenza è chiaramente la maggiore latenza. Analisi empiriche mostrano che la differenza è stimabile in termini di 4 o 5 ordini di grandezza meno veloce rispetto a un accesso locale, comunque in un range ancora significativamente più veloce di un accesso ad DBMS. Questo leggero incremento della latenza non è assolutamente un problema nella stragrande maggioranza dei casi, fatta eccezione per i contesti dove è necessario operare a bassa/bassissima latenza (pochi millisecondi) e operazioni batch che devono eseguire determinati servizi

su un ampio insieme di dati in un lasso di tempo relativamente piccolo.

Questi scenari possono essere affrontati in diversi modi, ad esempio: garantire che le richieste siano dirette ai nodi che memorizzano gli oggetti, precaricando specifici nodi con tutto l'insieme di oggetti che il processo richiede, distribuendo lo stesso processo di tutti i nodi, e così via.

Replicated cache, Local Cache, Near Cache

Una **cache replicata** (**replicated cache**) è un cluster **fault tolerant** in cui i dati sono completamente replicati per ciascun membro del cluster. Considerando il diagramma di figura 3, la cache replicata farebbe sì che entrambi i nodi abbiano tutti i quattro oggetti: **A**, **B**, **C** e **D**. Ciò significa che tutte le operazioni **get** sono risolte a livello locale, quindi questa opzione presenta la minore latenza in lettura (è la più veloce), a spese però della scalabilità e delle prestazioni degli aggiornamenti. Ogni scrittura deve essere elaborata da tutti i membri del cluster. Questo spiega il nome: una cache che replica i dati a tutti i nodi del cluster.

Cache locale (**local cache**) è in qualche modo l'opposto della cache replicata: una cache che è completamente contenuta all'interno di un particolare nodo di cluster (quindi locale). Ciò significa che non vi è un cluster e pertanto che esso **non** fornisce alcuna **tolleranza** al possibile fallimento del nodo.

La **cache vicina** (**near cache**) è una soluzione ibrida progettata per offrire il meglio dei due mondi: le superiori prestazioni della cache replicata e l'estrema scalabilità della cache locale fornendo rapido accesso in lettura ai dati **MRU** (utilizzati più recentemente) e MFU (utilizzati più di frequente). Pertanto, la **cache vicina** è un'implementazione che unisce due stili di cache: accesso cache locale (chiamato **cache front**) e una cache centralizzata (detta anche **a più livelli**) che può essere caricata su richiesta in caso di cache locale mancante (chiamata "cache indietro") .

Cache-Aside, Cache-Through e Write-Behind

Una **cache distribuita** può essere integrata nell'architettura di un sistema secondo diverse modalità di funzionamento, riconducibili ai tre seguenti pattern che prendono il nome dal posizionamento della cache nell'architettura: **Cache-Aside** (cache "affiancata"), **Cache-Through** (cache "attraverso") e Write-Behind ("aggiornamenti posticipati").

Cache-Aside

Cache-Aside, come il nome suggerisce, rappresenta un'architettura in cui la cache **affianca** il sistema esistente (figura 5) e pertanto si tratta di una soluzione intrinsecamente non intrusiva che si presta a essere implementata in termini di **cross-cutting concern** (in termini **AOP**).

Figura 5 – Struttura logica della Cache-Aside.

Lo sviluppatore si deve far carico di scrivere il codice necessario per gestire la **sincronizzazione** tra cache e sorgenti dati: per questo motivo nel **layer business object** sono presenti i **DAO** per accedere al database e per integrare la cache. Ciò impone che la cache e il database siano trattati come due componenti separati di un medesimo processo gestito dal **transaction manger**. In particolare, in lettura l'applicazione deve controllare la presenza in cache dei dati voluti e, in caso di esito negativo, procedere accedendo al database per poi memorizzarli nella cache. In scrittura, si deve occupare di aggiornare i dati nella cache durante l'aggiornamento del database.

Cache-Through

La soluzione Cache-Through prevede che la cache sia interposta tra la sorgente dati e l'applicazione (Figura 6), e pertanto ogni accesso alla sorgente dati deve passare attraverso la cache. In particolare, quando l'applicazione aggiorna un oggetto nella cache, l'operazione viene completata (e quindi la chiamata viene sbloccata) solo quando l'invocazione a Coherence è passata per il **CacheStore** che si occupa di memorizzare in modo sincrono l'oggetto, opportunamente convertito, nel database sottostante.

Figura 6 – Struttura logica della Cache-Through.

La scrittura **sincrona** rende la soluzione più immediata: per esempio, se la scrittura su database fallisce è immediato eseguire il roll-back. Ciò va però a discapito delle performance. In lettura, avviene una situazione analoga: qualora l'oggetto non sia presente in cache il **CacheStore** si occupa di recuperarlo dalla sorgente dati. In questa strategia, l'utilizzo della cache e delle relative librerie è alquanto pervasivo, quindi si ha relativamente meno codice da scrivere a fronte di una maggiore intrusività.

Inoltre, come nel caso di **Cache-Aside**, questa soluzione permette di aumentare le prestazioni e contestualmente di alleviare il carico di lavoro del database esclusivamente per quanto attiene alle operazioni di lettura, mentre le operazioni di scrittura continuano a essere completamente a carico della sorgente dati che tuttavia tende ad operare più agevolmente in quanto sgravato dal carico di lavoro dettato dalle query.

Soluzioni di tipo **Cache-Aside/-Through** tendono a portare un guadagno contenuto in sistemi che hanno la responsabilità di eseguire lunghi processi di fine giornata (**EOD**, End of Day) caratterizzati dal dover

processare grandi quantità di dati e dal dover aggiornare lo stato di ogni singolo record/oggetto. In questi, qualora si avesse bisogno di maggiori performance e/o di sgravare il database da una quantità eccessiva di lavoro, è opportuno valutare strategie come il **Write-Behind** menzionato di seguito.

Write-Behind

Write-Behind è una soluzione che dal punto di vista del funzionamento di differenzia significativamente dalle precedenti, in quanto si occupa anche di migliorare le prestazioni delle operazioni di aggiornamento. L'idea principale è di separare in termini di temporali (e quindi di transazionalità) l'aggiornamento della cache dal corrispondente aggiornamento del database. In altre parole, gli oggetti modificati in cache sono scritti in modo **asincrono** sulla base di dati dopo un **ritardo pre-configurato**. Coherence gestisce una coda write-behind dei dati che devono essere aggiornati nel database.

Figura 7 – Struttura logica Write-Behind: si è utilizzata una diversa rappresentazione, in quando la struttura è simile a quella del Cache-Aside.

Quando l'applicazione aggiorna l'**oggetto D** nella cache (figura 7), Coherence si occupa di "decorare" opportunamente l'oggetto: gli viene aggiunto un **flag** chiamato **store decoration**, che indica se è stato persistito o meno. L'oggetto decorato con il flag viene poi aggiunto alla coda **Write-Behind**, se non è già presente, altrimenti viene sostituito. Il nuovo oggetto (**cache entry**), come da prassi, è memorizzato in un **altro nodo** di backup. Eseguita questa operazione, la chiamata originaria è

libera di proseguire. Dopo un predefinito arco temporale, il **cache entry** diviene "maturo" (**ripe**) quindi il **CacheStore** è invocato per permettergli di aggiornare la sorgente dati sottostante con l'ultimo stato dell'**oggetto D**. A questo punto l'oggetto viene privato della decorazione.

Gli scenari che permettono alle strategie **Write-Behind** di fornire soluzioni veramente superiori sono quelli caratterizzati dal fatto che un medesimo oggetto possa variare diverse volte durante una finestra temporale relativamente piccola e l'applicazione abbia interesse a mantenere solo lo stato più aggiornato.

La soluzione di **scrittura posticipata** tende a portare significativi miglioramenti anche alle operazioni di aggiornamento in quanto non è più necessario attendere che i dati siano scritti nel database per poter procedere oltre e aggiornamenti multipli della cache possono essere combinati in una sola transazione sul database (**write-combining**) utilizzando il metodo **CacheStore.storeAll()** trasformando scritture singole in aggiornamenti batch. Tuttavia, come spesso accade, nulla è gratis ("there is no free supper"), e in effetti questa soluzione presenta un maggiore livello di complessità architetturale. In particolare, presenta tutta una serie di nuovi scenari da dover gestire attentamente, inerenti soprattutto l'**ordine di aggiornamento**, il **fallimento di scrittura** e il **rischio di perdita dati**.

Ordine di aggiornamento. Non è infrequente il caso in cui una medesima transazione richieda di aggiornare diversi grafi di oggetti, legati da opportune chiavi relazionali, in diverse **NamedCache** (mappe Oracle Coherence). Con la soluzione **Write-Behind** può accadere lo scenario in cui Coherence tenti di salvare nel database tali grafi di oggetti in un ordine diverso finendo per generare un'eccezione per violazione del vincolo di riferimento (**integrity reference**).

Fallimento di scrittura. Questo scenario accade quando per qualche motivo un oggetto memorizzato nella cache non riesce a essere persistito nel database sottostante. Una volta che l'oggetto è stato accettato nella cache, non è più possibile eseguire il **roll-back** della

transazione iniziale che ha determinato la memorizzazione. Quindi, con una codifica non accorta, si corre il rischio di avere oggetti nella cache che non si riesce a persistere definitivamente nel database.

Rischio di perdita di dati. Questo è spesso un rischio più teorico che reale, ma c'è sempre la possibilità che un intero cluster fallisca portando alla perdita irreversibile di dati.

Scalabilità lineare

Come descritto nel paragrafo di illustrazione di Coherence, il grid di Oracle è in grado di **scalare** in maniera **lineare**. Ciò significa che l'aggiunta di un nuovo nodo a un cluster aumenta effettivamente la potenza di calcolo (CPU), la capacità di memorizzazione di dati e, inevitabilmente, anche l'occupazione di banda. Tuttavia, ci sono delle importanti considerazioni da tenere a mente.

La strategia adottata da Coherence è un'implementazione del concetto di **Share-Nothing architecture** ("architettura a zero condivisione"): tutti i nodi sono indipendenti, autosufficienti e non esiste un punto di contesa che finisce per diventare un collo di bottiglia e limitare la scalabilità. Questo stile determina che letture dirette (**myCache.get(key)**) e scritture (**myCache.put(key, value)**) riescano effettivamente a trarre vantaggio dall'aumento del numero dei nodi del cluster. Ciò perche' il **load-balance** implicito utilizzato, basato sulla **hash** della chiave, fa sì che i dati siano effettivamente partizionati su tutti i sever disponibili e ogni singola operazione di lettura o scrittura possa essere indirizzata direttamente al solo nodo proprietario dell'oggetto.

L'operazione è quindi gestita da un solo server, a parte quello in cui ha origine la richiesta. Da tener presente però che in situazioni reali, semplici operazioni di **get** e **put** raramente risultano sufficienti. È poco realistico pensare di trovarsi ad implementare use case che gestiscano dati solo attraverso la chiave primaria. Molto più frequente è il caso in cui diversi servizi necessitino di utilizzare pattern di accesso complessi sul modello SQL query che inevitabilmente, non scalano altrettanto bene (la query

deve essere eseguita su diversi nodi e quindi anche i dati individuati risiedono su diversi nodi). Chiaramente, questi poblemi sono rilevabili solo in situazione estreme in termini di requisiti funzionali e di moli di dati da gestire. In questi casi, è necessario conoscere in dettaglio il funzionamento di Coherence e prendere le opportune precazioni come descritto in precedenza (fine paragrafo “Come funziona”).

Conclusioni

Con questo articolo abbiamo avviato il nostro viaggio all’interno della cache distribuita Oracle Coherence. L’obiettivo è di fornire una buona panoramica che consenta di comprendere dettagli tecnici presentati nel corso del prossimo articolo, con particolare riferimento a scelte architetturali e di disegno mirate ad implementare il compromesso AC del teorema CAP.

Le cache distribuite sono strumenti molto potenti che permettono di risolvere problemi legati al tradizionale collo di bottiglia generato dalla presenza di un database a cui tutti i servizi accedono. Anche se soluzioni RDBMS si sono evolute notevolmente nel tempo, restano fondamentalmente un servizio centralizzato non sempre adatto per la gestione di grandi volumi di dati maneggiati dai sistemi attuali di grandi organizzazioni come le banche. Detto questo, va notato che sebbene soluzioni grid siano in grado di dar luogo ad infrastrutture più flessibili, leggere (in termini di scalabilità) e potenti, finiscono per presentare tutta una nuova serie di sfide: decisione oculata sul numero di nodi, modalità di aggiornamento, bilanciamento tra performance e resilienza, topografia dei cluster, gestione delle eccezione in configurazioni write-behind, e così via.

Nel corso del prossimo articolo, presentiamo una serie di dettagli tecnici di basso livello. Per la maggior parte delle applicazione è possibile trascurare il funzionamento interno di Oracle Coherence: tuttavia questo non è il caso in presenza di requisiti funzionali particolarmente stringenti come per esempio latenze dell’ordine di uno-due millisecondi o necessità

di processare larghe quantità di dati.

Riferimenti

[1] Eric A. Brewer, ACM Symposium on the Principles of Distributed Computing

http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf

[2] Oracle e Tangosol

http://www.oracle.com/us/corporate/acquisitions/tangosol/index.html

[3] Luca Vetti Tagliati, “Component based architectures for eXtreme Transaction Processing”, SEKE 2008 Proceedings, July 1, 2008

www.ksi.edu/seke/Proceedings/seke/SEKE2008_Proceedings.pdf

[4] Noah Arliss – Joseph Ruzzi, “The Portable Object Format”, giugno 2009

http://coherence.oracle.com/display/COH35UG/The+Portable+Object+Format

[5] Luca Vetti Tagliati, “Java Best Practice – I migliori consigli per scrivere codice di qualità”, Tecniche Nuove, 2008

https://www.mokabyte.it/cms/article.run?articleId=2QE-UUY-MHK-EMB_7f000001_13985019_66035745