Ottimizzazione sul Cloud AWS di un document management system basato su MongoDB

17 Settembre 2021 - 5 min. read

In questi anni una moltitudine di applicazioni sono sviluppate in ottica Cloud-native: una architettura ben progettata non è però l'unico aspetto da considerare per arrivare al successo.

Nell'articolo di oggi vedremo come dedicare la giusta quantità di tempo all'esplorazione delle tecnologie cloud-native sia fondamentale per trovare la soluzione migliore, anche nel caso in cui scegliere un servizio al posto di un altro può implicare un refactor dell'applicazione.

Lo storage è un esempio che calza a pennello per la nostra discussione: oggi è disponibile una gamma vastissima di servizi di memorizzazione e archiviazione dei dati e utilizzare quello più adatto può essere la chiave giusta per riuscire a ridurre le attività di manutenzione e tenere bassi i costi dell'intera applicazione

Alcuni mesi fa ci è stato chiesto di migliorare le performance e ridurre i costi di un software per la memorizzazione di documenti. L'applicazione, basata su una architettura a microservizi e già in esecuzione su AWS, era stata sviluppata per essere ospitata sia On-Prem, che in Cloud e progettata per fornire alta affidabilità e scalabilità. Sul Cloud di AWS un cluster ECS Fargate era utilizzato per l'esecuzione dei container in autoscaling, mentre un cluster MongoDB Atlas con tre nodi dedicati alla memorizzazione dei documenti.

Gli oggetti erano memorizzati in formato BSON. Essendo in previsione la gestione e la memorizzazione di una enorme quantità di oggetti, l'elasticità è stata uno dei fattori determinanti per l'adozione del Cloud.

La crescita del numero di documenti gestiti ad alcuni milioni ha presto causato un notevole aumento dei costi, specialmente per lo storage e il traffico tra le diverse Availability Zone.

I costi per il trasferimento dati erano dovuti alla sincronizzazione del cluster: al fine di mantenere l'alta affidabilità, le istanze contenenti i dati erano in esecuzione in AZ differenti, per cui gli addebiti riflettevano la quantità di traffico di rete che il cluster doveva mantenere per la replica e la sincronizzazione.

Inoltre, i costi per lo Storage sono proporzionali alla dimensione dei volumi EBS necessari per la memorizzazione ed al parametro "replicationSpecs" di MongoDB.

In aggiunta è stato necessario procedere con il deploy di due nodi aggiuntivi per mantenere adeguati i livelli di performance durante i picchi di traffico.

Un ulteriore problema era dato dal lavoro richiesto per il mantenimento dei livelli di servizio durante le finestre di manutenzione o in caso di fallimento di un nodo del cluster.

Andava quindi trovata una strategia per ridurre i costi ed il lavoro necessario.

A volte la soluzione migliore ad un problema complesso è anche la più semplice: abbiamo deciso di utilizzare Amazon S3, uno dei primi servizi disponibili sul cloud AWS. Amazon S3 è uno storage ad oggetti progettato per offrire performance, alta disponibilità (con gli ormai famosi "undici 9" di affidabilità) e scalabilità. Mette a disposizione una ampia varietà di classi di storage con un ottimo rapporto qualità/prezzo ed API per la gestione dei dati che sono diventate lo standard de-facto.

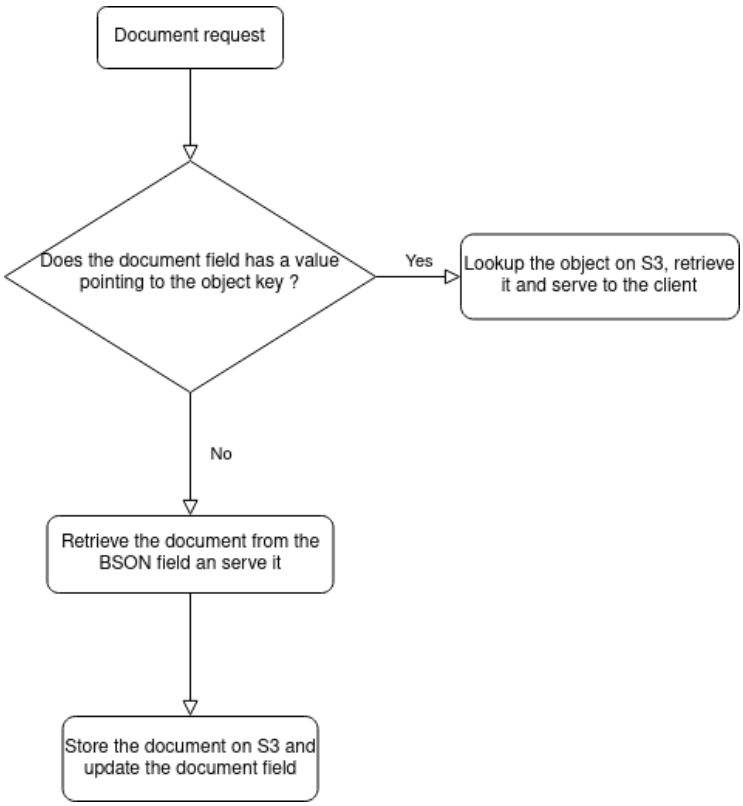
La scelta è caduta proprio su S3 perché:

- Rispetto ad EBS ha un migliore rapporto GB/prezzo (specialmente per alcune classi di archiviazione)
- I dati sono replicati tra più AZ
- I dati possono essere categorizzati da un sistema di "Intelligent Tiering" automatico che permette di abbassare i costi spostando i dati fra classi di storage a seconda della loro frequenza di utilizzo
- Impostare un sistema di replica tra più Region AWS è estremamente facile ed aiuta per il disaster recovery

Dopo aver sviluppato un prototipo, l'applicazione era pronta per utilizzare S3 per memorizzare i documenti. Era però necessario migrare i dati causando il disservizio più breve (e ragionevole) possibile.

Chiudere al pubblico l'applicazione, eseguire uno script per migrare i dati e rilasciare una nuova versione sarebbe stato troppo rischioso. In aggiunta, la mole di documenti era troppo grossa per poter terminare le operazioni in breve tempo.

L'approccio andava ripensato, spostando la responsabilità della migrazione all'applicazione stessa. Dopo alcune sessioni di brainstorming e design è emerso un piano molto più sicuro, che non richiedeva alcuna interruzione di servizio. Con un piccolo refactor dell'applicazione la logica di recupero dei documenti è stata modificata aggiungendo un nuovo campo a database contenente l'identificatore dell'oggetto memorizzato nel bucket e modificando poi il flusso di lettura del documento secondo questo schema logico:



Nonostante questa implementazione, era comunque necessario completare in un tempo ragionevole la migrazione dei dati per procedere al ridimensionamento del cluster MongoDB ed assicurarsi che tutti i documenti fossero memorizzati su S3.

Per fare questo abbiamo sviluppato uno script python che sfruttasse la logica applicativa per migrare i dati. In questo modo, una semplice richiesta ad un documento era sufficiente a scatenare la migrazione.

Abbiamo iniziato generando le liste di documenti da migrare. Ciascuna lista poteva contenere fino a 70 milioni di oggetti, per cui è stato necessario creare piccoli batch per rendere le operazioni parallelizzabili e gestibili in caso di fallimento.

Per memorizzare lo stato di migrazione di ogni oggetto e procedere alle verifiche post-migrazione era necessario un database ad alte performance e che fosse in grado di gestire milioni di record di tipo chiave-valore. La nostra scelta è caduta (ovviamente) su DynamoDB. Abbiamo configurato quindi una tabella contenente l'identificatore dell'oggetto (usato come chiave), lo stato della migrazione ed un hash del documento richiesto.

Per l'esecuzione dello script abbiamo scelto di utilizzare istanze EC2: un primo test su una istanza t3 ha evidenziato la necessità di avere più memoria per riuscire ad eseguire la migrazione in parallelo. Abbiamo quindi utilizzato la famiglia m5, riuscendo a parallelizzare ulteriormente le operazioni.

Per gestire i fallimenti abbiamo implementato una logica che controllasse lo stato di migrazione sulla tabella DynamoDB prima di inoltrare la richiesta all'applicazione, in modo da eliminare eventuali chiamate non necessarie.

Terminata la migrazione ci siamo assicurati che nessun dato fosse corrotto eseguendo uno script aggiuntivo per calcolare l'hash di tutti gli oggetti presenti su S3 e comparandolo con quelli memorizzati su DynamoDB.

Dopo un mese sono stati migrati con successo tutti i 40 Terabyte di dati senza alcun impatto sugli utenti e sulla disponibilità del servizio. La riduzione significativa di costi e tempi ha permesso al business di espandersi, acquisendo più utenti e potenza di fuoco.

Conclusioni

Il Cloud non è una bacchetta magica che risolve ogni problema. Anche se le applicazioni sembrano funzionare perfettamente una volta portate in Cloud, spesso occorre ripensarle in parte o nella loro totalità affinché funzionino efficentemente e siano ottimizzate in termini di costi e performance.

E voi come avete ottimizzato le vostre applicazioni per il Cloud? Raccontatecelo nei commenti!

Damiano Giorgi

Ex sistemista on-prem, pigro e incline all'automazione di task noiosi. Alla ricerca costante di novità tecnologiche e quindi passato al cloud per trovare nuovi stimoli.L'unico hardware a cui mi dedico ora è quello del mio basso; se non mi trovate in ufficio o in sala prove provate al pub o in qualche aeroporto!
