



Le migliori pratiche per l'ottimizzazione delle prestazioni per i job di AWS Glue Apache Spark



: Le migliori pratiche per l'ottimizzazione delle prestazioni per i job di AWS Glue Apache Spark

Table of Contents

Introduzione	1
Argomenti chiave	2
Architecture	2
Set di dati distribuito resiliente	3
Valutazione pigra	5
Terminologia delle applicazioni Spark	6
Parallelism	7
Ottimizzatore Catalyst	8
Analizza i problemi di prestazioni	11
Identifica i punti deboli utilizzando l'interfaccia utente di Spark	11
Strategie per ottimizzare le prestazioni	13
Strategia di base per l'ottimizzazione delle prestazioni	13
Pratiche di ottimizzazione per le prestazioni lavorative di Spark	14
Scalare la capacità del cluster	15
CloudWatch metriche	15
Interfaccia utente Spark	16
Usa la versione più recente	17
Ridurre la quantità di scansione dei dati	18
CloudWatch metriche	18
Interfaccia utente Spark	19
Parallelizza le attività	27
CloudWatch metriche	28
Interfaccia utente Spark	28
Ottimizza gli shuffles	34
CloudWatch metriche	35
Interfaccia utente Spark	35
Ridurre al minimo le spese generali di pianificazione	44
CloudWatch metriche	44
Interfaccia utente Spark	45
Ottimizza le funzioni definite dall'utente	46
UDF standard in Python	47
UDF vettorizzata	48
Spark SQL	49
Usare i panda per i big data	49

Risorse 50

Cronologia dei documenti 51

..... lii

Le migliori pratiche per l'ottimizzazione delle prestazioni per i job di AWS Glue Apache Spark

Roman Myers, Takashi Onikura e Noritaka Sekiyama, Amazon Web Services (AWS)

Dicembre 2023 ([cronologia dei documenti](#))

AWS Glue offre diverse opzioni per ottimizzare le prestazioni. Questa guida definisce gli argomenti chiave per l'ottimizzazione di Apache AWS Glue Spark. Fornisce quindi una strategia di base da seguire per ottimizzarli AWS Glue per i job di Apache Spark. Usa questa guida per imparare a identificare i problemi di prestazioni interpretando le metriche disponibili in AWS Glue. Quindi incorpora strategie per risolvere questi problemi, massimizzando le prestazioni e riducendo al minimo i costi.

Questa guida illustra le seguenti pratiche di ottimizzazione:

- [Scalare la capacità del cluster](#)
- [Usa la AWS Glue versione più recente](#)
- [Riduci la quantità di scansione dei dati](#)
- [Parallelizza le attività](#)
- [Riduci al minimo i costi di pianificazione](#)
- [Ottimizza gli shuffles](#)
- [Ottimizza le funzioni definite dall'utente](#)

Argomenti chiave di Apache Spark

Questa sezione spiega i concetti di base di Apache Spark e gli argomenti chiave per l'ottimizzazione delle prestazioni di Apache AWS Glue Spark. È importante comprendere questi concetti e argomenti prima di discutere delle strategie di ottimizzazione del mondo reale.

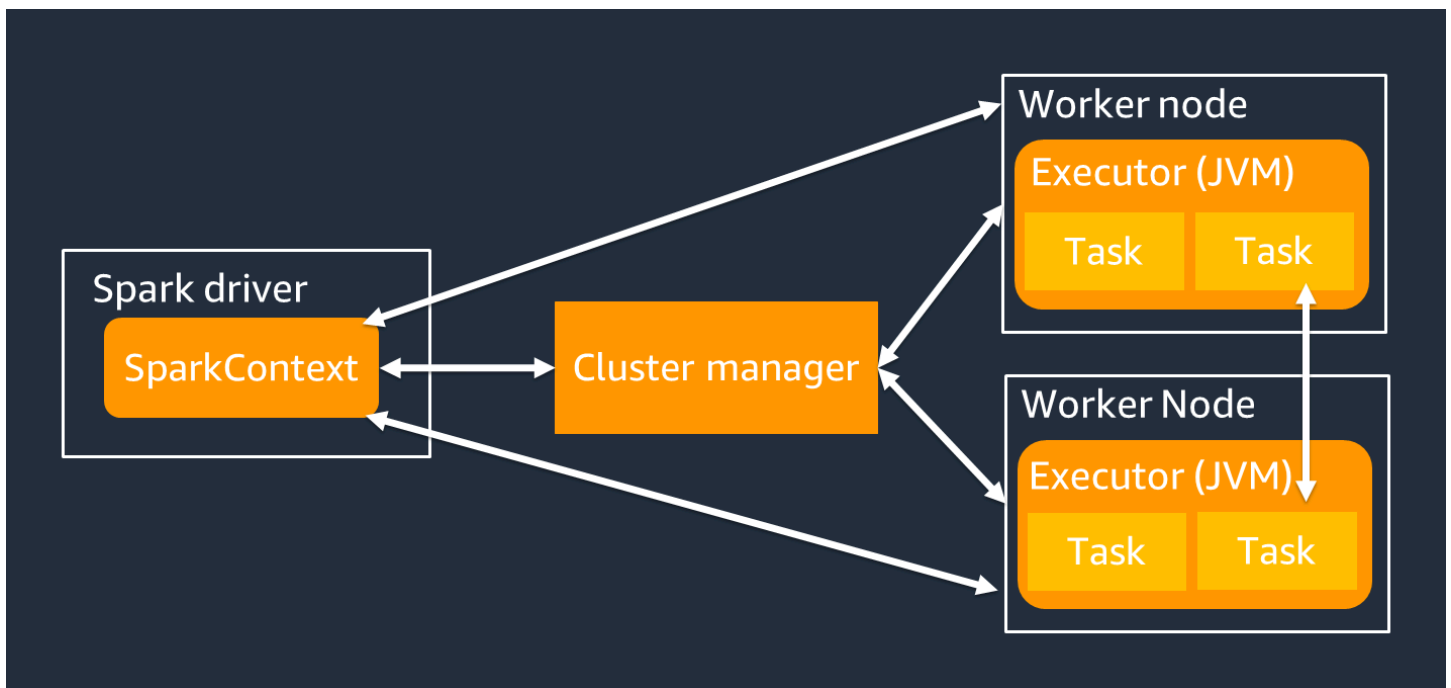
Architecture

Il driver Spark è principalmente responsabile della suddivisione dell'applicazione Spark in attività che possono essere eseguite su singoli lavoratori. Il driver Spark ha le seguenti responsabilità:

- È in esecuzione `main()` nel tuo codice
- Generazione di piani di esecuzione
- Fornitura degli esecutori Spark in collaborazione con il gestore del cluster, che gestisce le risorse del cluster
- Pianificazione delle attività e richiesta di attività per gli esecutori Spark
- Gestione dell'avanzamento e del ripristino delle attività

Si utilizza un `SparkContext` oggetto per interagire con il driver Spark durante l'esecuzione del lavoro.

Un esecutore Spark è un lavoratore che conserva dati ed esegue attività che vengono trasmesse dal driver Spark. Il numero di esecutori Spark aumenterà e diminuirà in base alle dimensioni del cluster.



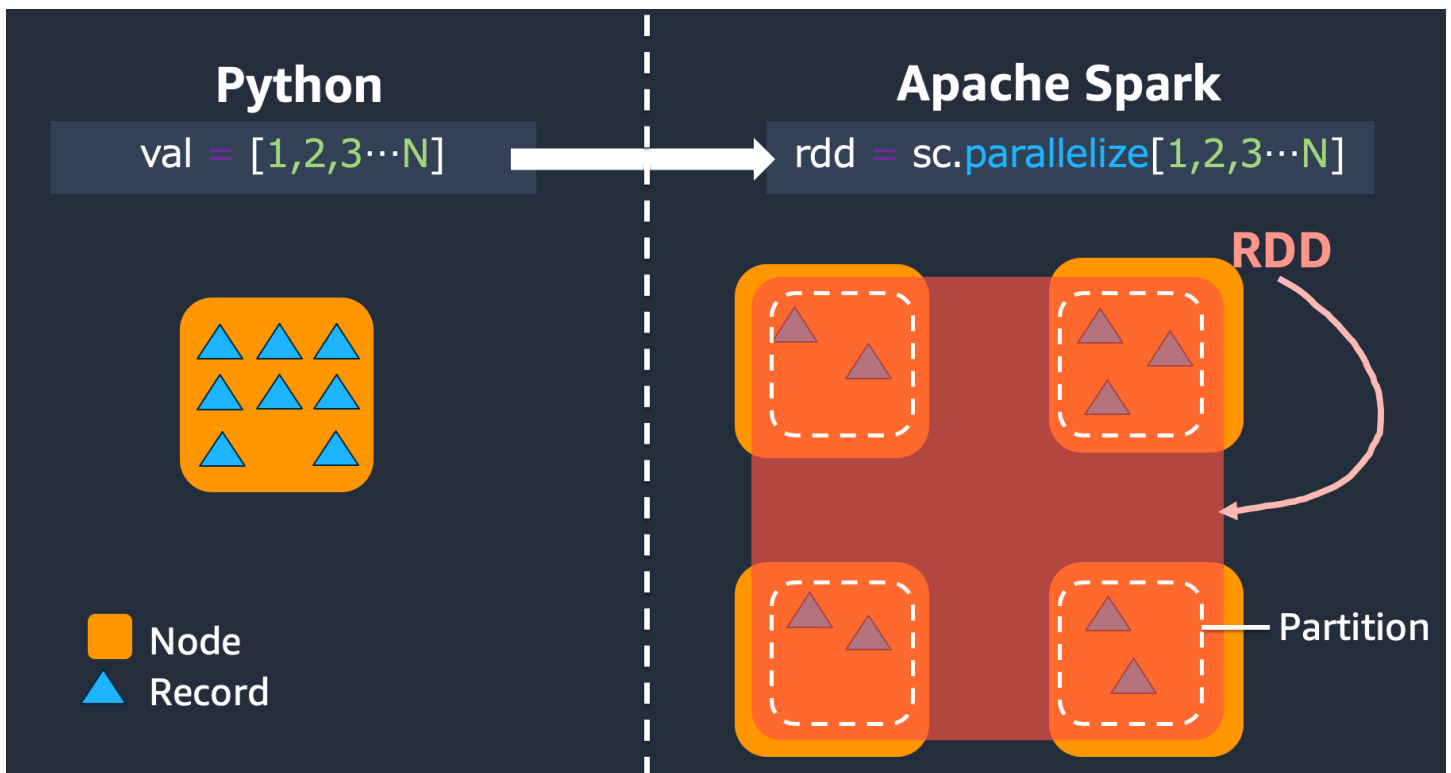
Note

Un esecutore Spark ha più slot in modo che più attività vengano elaborate in parallelo. Per impostazione predefinita, Spark supporta un'attività per ogni core della CPU virtuale (vCPU). Ad esempio, se un executor ha quattro core CPU, può eseguire quattro attività simultanee.

Set di dati distribuito resiliente

Spark svolge il complesso lavoro di archiviazione e tracciamento di set di dati di grandi dimensioni tra gli esecutori Spark. Quando scrivi codice per i job Spark, non devi pensare ai dettagli dello storage. Spark fornisce l'astrazione resiliente del set di dati distribuito (RDD), che è una raccolta di elementi su cui è possibile operare in parallelo e che possono essere partizionati tra gli esecutori Spark del cluster.

La figura seguente mostra la differenza nel modo in cui archiviare i dati in memoria quando uno script Python viene eseguito nel suo ambiente tipico e quando viene eseguito nel framework Spark ().
PySpark



- Python: la scrittura `val = [1, 2, 3...N]` in uno script Python mantiene i dati in memoria sulla singola macchina su cui è in esecuzione il codice.
- PySpark— Spark fornisce la struttura dati RDD per caricare ed elaborare i dati distribuiti nella memoria su più esecutori Spark. Puoi generare un RDD con codice come `rdd = sc.parallelize[1, 2, 3...N]`, e Spark può distribuire e conservare automaticamente i dati in memoria su più esecutori Spark.

In molti AWS Glue lavori, usi RDD tramite e Spark. AWS Glue `DynamicFrames`/`DataFrames` Si tratta di astrazioni che consentono di definire lo schema dei dati in un RDD ed eseguire attività di livello superiore con tali informazioni aggiuntive. Poiché utilizzano RDD internamente, i dati vengono distribuiti in modo trasparente e caricati su più nodi nel seguente codice:

- `DynamicFrame`

```
dyf= glueContext.create_dynamic_frame.from_options(  
    's3', {"paths": [ "s3://<YourBucket>/<Prefix>/" ]},  
    format="parquet",  
    transformation_ctx="dyf"  
)
```

- `DataFrame`

```
df = spark.read.format("parquet")  
    .load("s3://<YourBucket>/<Prefix>")
```

Un RDD presenta le seguenti funzionalità:

- Gli RDD sono costituiti da dati suddivisi in più parti chiamate partizioni. Ogni esecutore Spark archivia una o più partizioni in memoria e i dati vengono distribuiti su più esecutori.
- Gli RDD sono immutabili, il che significa che non possono essere modificati dopo essere stati creati. Per modificare un DataFrame, puoi usare le trasformazioni, che sono definite nella sezione seguente.
- Gli RDD replicano i dati tra i nodi disponibili, in modo da poter eseguire automaticamente il ripristino in caso di errori dei nodi.

Valutazione pigra

Gli RDD supportano due tipi di operazioni: le trasformazioni, che creano un nuovo set di dati da uno esistente, e le azioni, che restituiscono un valore al programma driver dopo aver eseguito un calcolo sul set di dati.

- Trasformazioni: poiché gli RDD sono immutabili, è possibile modificarli solo utilizzando una trasformazione.

Ad esempio, map è una trasformazione che passa ogni elemento del set di dati attraverso una funzione e restituisce un nuovo RDD che rappresenta i risultati. Notate che il map metodo non restituisce un output. Spark memorizza la trasformazione astratta per il futuro, invece di lasciarti interagire con il risultato. Spark non agirà sulle trasformazioni finché non richiederai un'azione.

- Azioni: utilizzando le trasformazioni, costruisci il tuo piano di trasformazione logica. Per avviare il calcolo, si esegue un'azione come write, count, show o collect

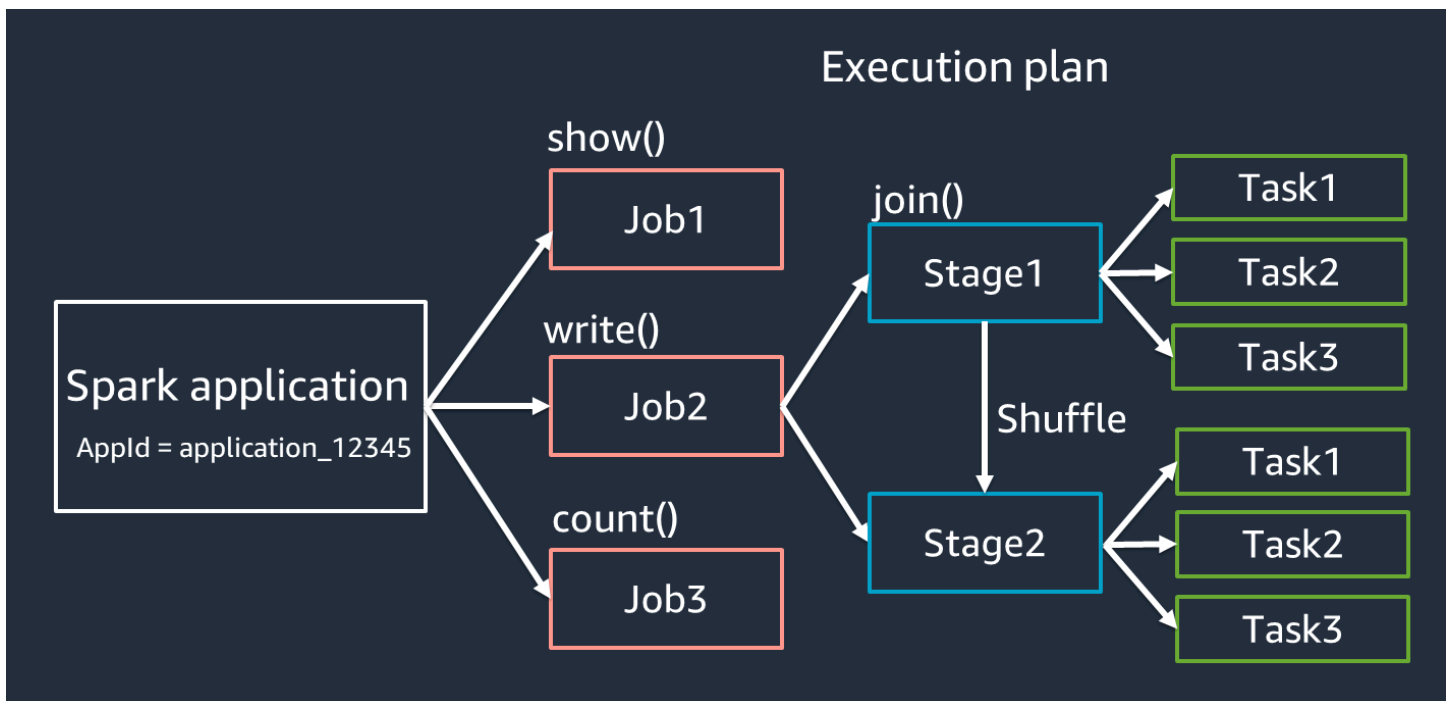
Tutte le trasformazioni in Spark sono lente, in quanto non calcolano immediatamente i risultati. Spark ricorda invece una serie di trasformazioni applicate ad alcuni set di dati di base, come gli oggetti Amazon Simple Storage Service (Amazon S3). Le trasformazioni vengono calcolate solo quando un'azione richiede la restituzione di un risultato al driver. Questo design consente a Spark di funzionare in modo più efficiente. Ad esempio, si consideri la situazione in cui un set di dati creato tramite la map trasformazione viene utilizzato solo da una trasformazione che riduce

sostanzialmente il numero di righe, ad esempio. `reduce` È quindi possibile passare al driver il set di dati più piccolo che ha subito entrambe le trasformazioni, invece di passare il set di dati mappato più grande.

Terminologia delle applicazioni Spark

Questa sezione tratta la terminologia delle applicazioni Spark. Il driver Spark crea un piano di esecuzione e controlla il comportamento delle applicazioni in diverse astrazioni. I seguenti termini sono importanti per lo sviluppo, il debug e l'ottimizzazione delle prestazioni con l'interfaccia utente Spark.

- Applicazione: basata su una sessione Spark (contesto Spark). Identificato da un ID univoco come `<application_XXX>`
- Offerte di lavoro: in base alle azioni create per un RDD. Un lavoro è costituito da una o più fasi.
- Fasi: basato sugli shuffle creati per un RDD. Una fase è costituita da una o più attività. Lo shuffle è il meccanismo di Spark per ridistribuire i dati in modo che siano raggruppati in modo diverso tra le partizioni RDD. Alcune trasformazioni, ad esempio, richiedono un shuffle. `join()` [Lo shuffle viene discusso più dettagliatamente nella pratica di ottimizzazione di `Optimize shuffles`.](#)
- Attività: un'attività è l'unità minima di elaborazione pianificata da Spark. Le attività vengono create per ogni partizione RDD e il numero di attività è il numero massimo di esecuzioni simultanee nella fase.



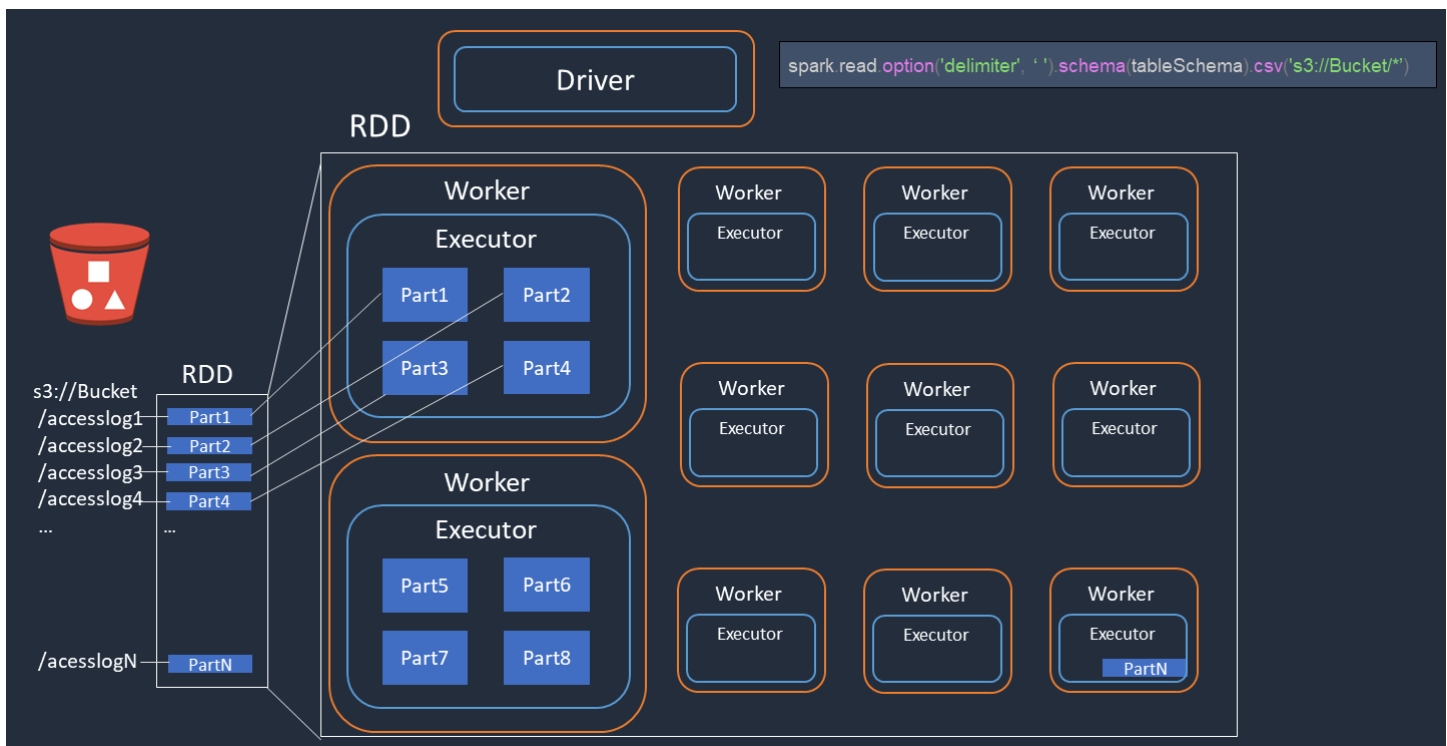
Note

Le attività sono la cosa più importante da considerare quando si ottimizza il parallelismo. Il numero di attività varia in base al numero di RDD

Parallelism

Spark parallelizza le attività per il caricamento e la trasformazione dei dati.

Prendi in considerazione un esempio in cui esegui l'elaborazione distribuita dei file di log di accesso (denominati `accesslog1` ... `accesslogN`) su Amazon S3. Il diagramma seguente mostra il flusso di elaborazione distribuito.

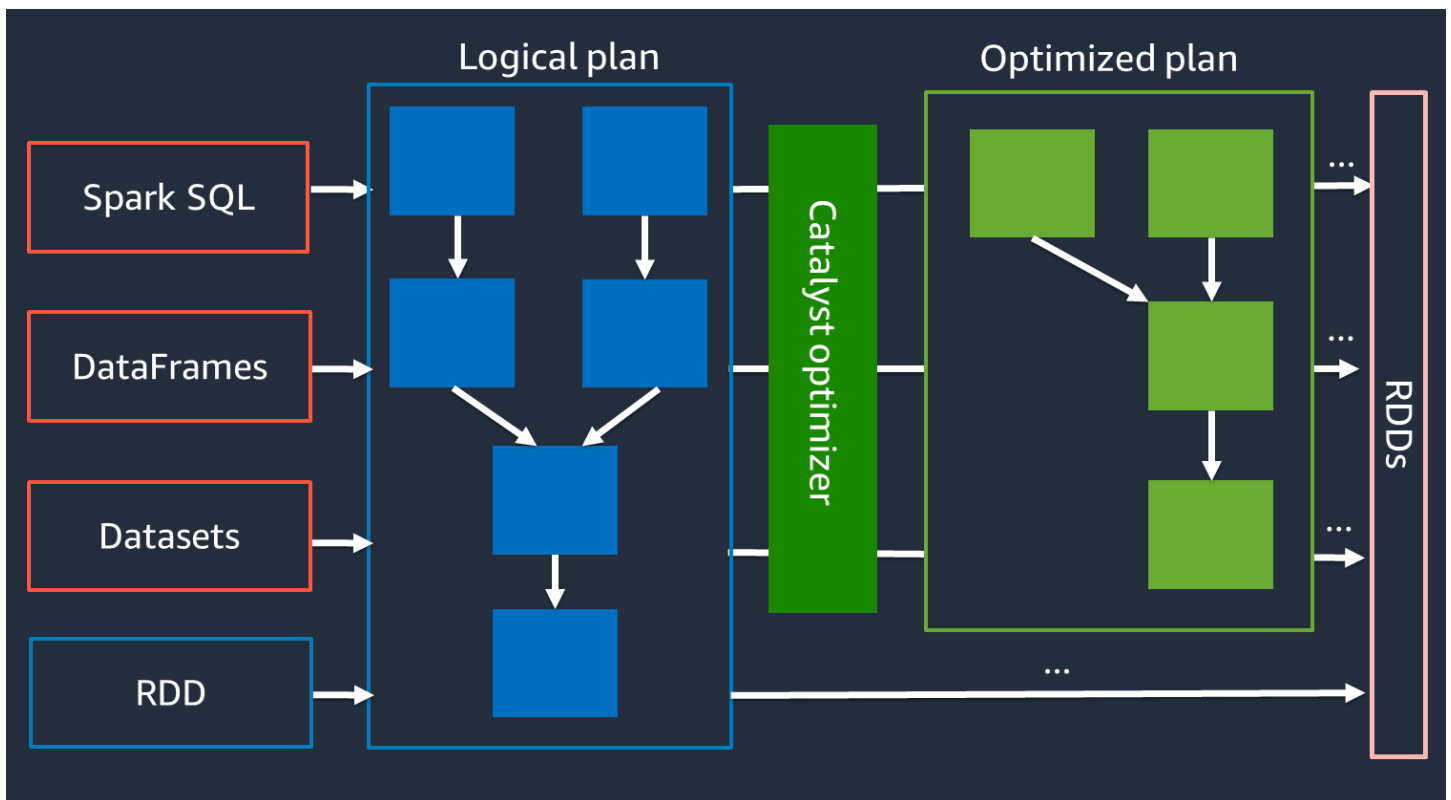


1. Il driver Spark crea un piano di esecuzione per l'elaborazione distribuita su molti esecutori Spark.
2. Il driver Spark assegna le attività a ciascun esecutore in base al piano di esecuzione. Per impostazione predefinita, il driver Spark crea partizioni RDD (ciascuna corrispondente a un'attività Spark) per ogni oggetto S3 (). Part1 . . . N Quindi il driver Spark assegna le attività a ciascun esecutore.
3. Ogni task Spark scarica l'oggetto S3 assegnato e lo archivia in memoria nella partizione RDD. In questo modo, più esecutori Spark scaricano ed elaborano l'attività assegnata in parallelo.

Per maggiori dettagli sul numero iniziale di partizioni e sull'ottimizzazione, consulta la sezione [Parallelizza](#) le attività.

Ottimizzatore Catalyst

Internamente, Spark utilizza un motore chiamato [Catalyst](#) Optimizer per ottimizzare i piani di esecuzione. [Catalyst dispone di un ottimizzatore di query che puoi usare quando esegui API Spark di alto livello, come Spark SQL e Datasets, come descritto nel diagramma seguente. DataFrame](#)



Poiché l'ottimizzatore Catalyst non funziona direttamente con l'API RDD, le API di alto livello sono generalmente più veloci delle API RDD di basso livello. Per i join complessi, l'ottimizzatore Catalyst può migliorare significativamente le prestazioni ottimizzando il piano di esecuzione dei job. Puoi vedere il piano ottimizzato del tuo job Spark nella scheda SQL dell'interfaccia utente di Spark.

Esecuzione adattiva delle interrogazioni

L'ottimizzatore Catalyst esegue l'ottimizzazione del runtime tramite un processo chiamato Adaptive Query Execution. Adaptive Query Execution utilizza le statistiche di runtime per ottimizzare nuovamente il piano di esecuzione delle query durante l'esecuzione del lavoro. Adaptive Query Execution offre diverse soluzioni ai problemi di prestazioni, tra cui la coalescenza delle partizioni post-shuffle, la conversione dello sort-merge join in broadcast join e l'ottimizzazione dello skew join, come descritto nelle sezioni seguenti.

Adaptive Query Execution è disponibile nella versione 3.0 e successive ed è abilitata per AWS Glue impostazione predefinita nella versione 4.0 (Spark 3.3.0) e versioni successive. AWS Glue Adaptive Query Execution può essere attivata e disattivata utilizzando `spark.conf.set("spark.sql.adaptive.enabled", "true")` il codice.

Partizioni unificate dopo lo shuffle

Questa funzione riduce le partizioni RDD (coalescenza) dopo ogni mescolamento in base alle statistiche di output. map Semplifica l'ottimizzazione del numero di partizione shuffle durante l'esecuzione di query. Non è necessario impostare un numero di partizione shuffle per adattarlo al set di dati. Spark può scegliere il numero di partizione shuffle corretto in fase di esecuzione dopo aver ottenuto un numero iniziale sufficientemente grande di partizioni shuffle.

La coalescenza delle partizioni post-shuffle è abilitata quando entrambe le partizioni sono impostate su true. `spark.sql.adaptive.enabled`
`spark.sql.adaptive.coalescePartitions.enabled` [Per ulteriori informazioni, consulta la documentazione di Apache Spark.](#)

Conversione da sort-merge join a broadcast join

Questa funzione riconosce quando si uniscono due set di dati di dimensioni sostanzialmente diverse e adotta un algoritmo di join più efficiente basato su tali informazioni. [Per maggiori dettagli, consulta la documentazione di Apache Spark.](#) Le strategie di unione sono discusse nella sezione [Optimize shuffles](#).

Ottimizzazione delle giunzioni asimmetriche

La distorsione dei dati è uno degli ostacoli più comuni per i lavori Spark. Descrive una situazione in cui i dati vengono distorti su partizioni RDD specifiche (e di conseguenza, attività specifiche), il che ritarda il tempo di elaborazione complessivo dell'applicazione. Ciò può spesso ridurre le prestazioni delle operazioni di join. La funzionalità di ottimizzazione dello skew join gestisce dinamicamente l'inclinazione nei join di tipo ordinamento e unione suddividendo (e replicando se necessario) le attività asimmetriche in attività di dimensioni approssimativamente uguali.

Questa `spark.sql.adaptive.skewJoin.enabled` funzionalità è abilitata quando è impostata su true. Per maggiori dettagli, consulta la documentazione di [Apache Spark](#). La distorsione dei dati viene discussa ulteriormente nella sezione [Optimize shuffles](#).

Analizza i problemi di prestazioni utilizzando l'interfaccia utente Spark

Prima di applicare le migliori pratiche per ottimizzare le prestazioni dei AWS Glue lavori, consigliamo vivamente di profilare le prestazioni e identificare gli ostacoli. Questo ti aiuterà a concentrarti sulle cose giuste.

Per un'analisi rapida, le [CloudWatch metriche di Amazon](#) forniscono una visualizzazione di base delle metriche relative alle tue offerte di lavoro. L'[interfaccia utente Spark](#) offre una visione più approfondita per l'ottimizzazione delle prestazioni. Per utilizzare l'interfaccia utente Spark con AWS Glue, devi [abilitare l'interfaccia utente Spark per i](#) tuoi lavori. AWS Glue Dopo aver acquisito dimestichezza con l'interfaccia utente di Spark, segui le [strategie per ottimizzare le prestazioni lavorative di Spark](#) per identificare e ridurre l'impatto delle strozzature sulla base delle tue scoperte.

Identifica i punti deboli utilizzando l'interfaccia utente di Spark

Quando apri l'interfaccia utente di Spark, le applicazioni Spark vengono elencate in una tabella. Per impostazione predefinita, il nome dell'app di un AWS Glue lavoro è. nativespark-<Job Name>-<Job Run ID> Scegli l'app Spark di destinazione in base al Job Run ID per aprire la scheda Jobs. Le esecuzioni di job incomplete, come le esecuzioni di job in streaming, sono elencate in Mostra applicazioni incomplete.

La scheda Lavori mostra un riepilogo di tutti i lavori nell'applicazione Spark. Per determinare eventuali fasi o errori delle attività, controlla il numero totale di attività. Per individuare i punti deboli, ordina scegliendo Durata. Approfondisci i dettagli dei lavori di lunga durata selezionando il link mostrato nella colonna Descrizione.

Spark Jobs (?)
 User: spark
 Total Uptime: 7.7 min
 Scheduling Mode: FIFO
 Completed Jobs: 7
 ▶ Event Timeline
 - Completed Jobs (7)

Page: 1
 1 Pages. Jump to 1. Show 100 items in a page. Go

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
3	parquet at NativeMethodAccessorImpl.java:0 parquet at NativeMethodAccessorImpl.java:0	2023/03/30 06:49:02	6.5 min	1/1 (1 skipped)	5/5 (799 skipped)
0	showString at NativeMethodAccessorImpl.java:0 showString at NativeMethodAccessorImpl.java:0	2023/03/30 06:48:15	29 s	1/1	799/799
2	parquet at NativeMethodAccessorImpl.java:0 parquet at NativeMethodAccessorImpl.java:0	2023/03/30 06:48:48	14 s	1/1	799/799

La pagina Details for Job elenca le fasi. In questa pagina puoi visualizzare informazioni generali come la durata, il numero di attività completate e totali, il numero di input e output e la quantità di letture e scritture casuali.

Details for Job 3

Status: SUCCEEDED

Submitted: 2023/03/30 06:49:02

Duration: 6.5 min

Associated SQL Query: 2

Completed Stages: 1

Skipped Stages: 1

Event Timeline

DAG Visualization

Completed Stages (1)

Page: 1

1 Pages. Jump to 1. Show 100 Items in a page. Go

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
5	parquet at NativeMethodAccessorImpl.java:0	+details 2023/03/30 06:49:02	6.5 min	5/5		10.2 GiB	11.9 GiB	

La scheda Executor mostra in dettaglio la capacità del cluster Spark. Puoi controllare il numero totale di core. Il cluster mostrato nella schermata seguente contiene 316 core attivi e 512 core in totale. Per impostazione predefinita, ogni core può elaborare un task Spark contemporaneamente.

Executors

Show Additional Metrics

Summary

	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks
Active(80)	0	0.0 B / 465.9 GiB	0.0 B	316	10	0	2399	2399
Dead(49)	0	0.0 B / 285.4 GiB	0.0 B	196	10	0	3	3
Total(129)	0	0.0 B / 751.3 GiB	0.0 B	512	10	0	2402	2402

In base al valore 5/5 mostrato nella pagina Details for Job, la fase 5 è la fase più lunga, ma utilizza solo 5 core su 512. Poiché il parallelismo in questa fase è così basso, ma richiede molto tempo, è possibile identificarla come un collo di bottiglia. Per migliorare le prestazioni, è necessario capire perché. Per saperne di più su come riconoscere e ridurre l'impatto dei comuni rallentamenti prestazionali, consulta Strategie per ottimizzare le prestazioni lavorative di Spark.

Strategie per ottimizzare le prestazioni lavorative di Spark

Quando ci si prepara all'ottimizzazione dei parametri, utilizza la seguenti best practice:

- Determinare gli obiettivi di prestazioni prima di iniziare a identificare i problemi.
- Prima di provare di modificare i parametri di ottimizzazione, utilizza i parametri per identificare i problemi.

Per ottenere risultati più coerenti durante l'ottimizzazione di un processo, sviluppa una strategia di base per il lavoro di ottimizzazione.

Strategia di base per l'ottimizzazione delle prestazioni

In genere, l'ottimizzazione delle prestazioni viene eseguita nel seguente flusso di lavoro:

1. Determinazione degli obiettivi delle prestazioni.
2. Misurazione dei parametri.
3. Identificazione dei colli di bottiglia.
4. Riduzione dell'impatto dei colli di bottiglia.
5. Ripeti i passaggi da 2 a 4 fino a raggiungere l'obiettivo prefissato.

Innanzitutto, stabilisci i tuoi obiettivi prestazionali. Ad esempio, uno dei tuoi obiettivi potrebbe essere quello di completare un AWS Glue lavoro entro 3 ore. Dopo aver definito i tuoi obiettivi, misura le metriche delle prestazioni lavorative. Identifica le tendenze nelle metriche e gli ostacoli per raggiungere gli obiettivi. In particolare, identificare i punti deboli è molto importante per la risoluzione dei problemi, il debug e l'ottimizzazione delle prestazioni. Durante l'esecuzione di un'applicazione Spark, Spark registra lo stato e le statistiche di ogni attività nel registro degli eventi Spark.

In AWS Glue, puoi visualizzare le metriche di Spark tramite l'[interfaccia utente Web di Spark](#) fornita dal server di cronologia Spark. AWS Glue for Spark jobs può inviare [i log degli eventi Spark](#) a una posizione specificata in Amazon S3. AWS Glue fornisce anche un [AWS CloudFormation modello](#) di esempio e un [Dockerfile](#) per avviare il server di cronologia Spark su un'istanza Amazon EC2 o sul computer locale, in modo da poter utilizzare l'interfaccia utente Spark con i registri degli eventi.

Dopo aver determinato i tuoi obiettivi prestazionali e identificato le metriche per valutarli, puoi iniziare a identificare e correggere gli ostacoli utilizzando le strategie illustrate nelle sezioni seguenti.

Pratiche di ottimizzazione per le prestazioni lavorative di Spark

Puoi utilizzare le seguenti strategie per ottimizzare le prestazioni dei job AWS Glue Spark:

- AWS Glue risorse:
 - [Scalare la capacità del cluster](#)
 - [Usa la versione più recente AWS Glue version](#)
- Applicazioni Spark:
 - [Ridurre la quantità di scansione dei dati](#)
 - [Parallelizza le attività](#)
 - [Ottimizza gli shuffles](#)
 - [Riduci al minimo i costi di pianificazione](#)
 - [Ottimizza le funzioni definite dall'utente](#)

Prima di utilizzare queste strategie, devi avere accesso alle metriche e alla configurazione per il tuo job Spark. [Puoi trovare queste informazioni nella documentazione.AWS Glue](#)

Dal punto di vista delle AWS Glue risorse, è possibile ottenere miglioramenti delle prestazioni aggiungendo AWS Glue lavoratori e utilizzando la AWS Glue versione più recente.

Dal punto di vista delle applicazioni Apache Spark, hai accesso a diverse strategie che possono migliorare le prestazioni. Se nel cluster Spark vengono caricati dati non necessari, puoi rimuoverli per ridurre la quantità di dati caricati. Se hai sottoutilizzato le risorse del cluster Spark e hai pochi dati I/O, puoi identificare le attività da parallelizzare. Potresti anche voler ottimizzare le operazioni pesanti di trasferimento dei dati, come i join, se richiedono molto tempo. Puoi anche ottimizzare il tuo piano di richieste di lavoro o ridurre la complessità computazionale delle singole attività Spark.

Per applicare in modo efficiente queste strategie, devi identificare quando sono applicabili consultando le tue metriche. Per ulteriori dettagli, consulta ciascuna delle seguenti sezioni. Queste tecniche sono utili non solo per ottimizzare le prestazioni, ma anche per risolvere problemi tipici come gli errori di esaurimento della memoria (OOM).

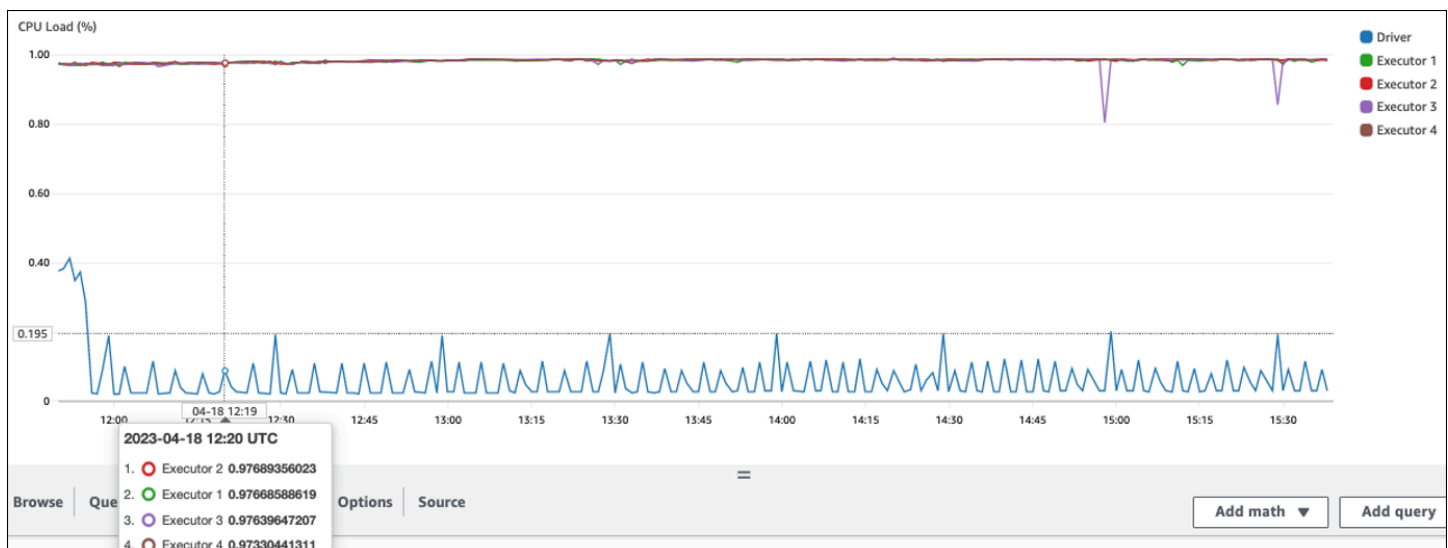
Scalare la capacità del cluster

Se il tuo lavoro richiede troppo tempo, ma gli esecutori consumano risorse sufficienti e Spark sta creando un grande volume di attività relative ai core disponibili, prendi in considerazione la possibilità di scalare la capacità del cluster. Per valutare se ciò è appropriato, utilizza le seguenti metriche.

CloudWatch metriche

- Controlla il carico della CPU e l'utilizzo della memoria per determinare se gli executor stanno consumando risorse sufficienti.
- Controllate la durata del processo per valutare se il tempo di elaborazione è troppo lungo per soddisfare gli obiettivi prestazionali.

Nell'esempio seguente, quattro executor sono in esecuzione con un carico della CPU superiore al 97%, ma l'elaborazione non è stata completata dopo circa tre ore.



Note

Se il carico della CPU è basso, probabilmente non trarrete alcun vantaggio dalla scalabilità della capacità del cluster.

Interfaccia utente Spark

Nella scheda Job o Stage, puoi vedere il numero di attività per ogni lavoro o fase. Nell'esempio seguente, Spark ha creato 58100 delle attività.

Stages for All Jobs					
Completed Stages: 1					
- Completed Stages (1)					
Stage Id ▾	Description		Submitted	Duration	Tasks: Succeeded/Total
0	count at DynamicFrame.scala:1414	+details	2023/04/18 10:59:10	4.8 h	58100/58100
					Input 28.4 GB

Nella scheda Esecutore, puoi vedere il numero totale di esecutori e attività. Nella schermata seguente, ogni esecutore Spark ha quattro core e può eseguire quattro attività contemporaneamente.

Executors						
Show	20 ▾	entries				
Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores
driver	172.35.229.149:37603	Active	0	0.0 B / 6.3 GB	0.0 B	0
1	172.34.249.100:34733	Active	0	0.0 B / 6.3 GB	0.0 B	4
2	172.35.72.25:38929	Active	0	0.0 B / 6.3 GB	0.0 B	4
3	172.34.49.138:39961	Active	0	0.0 B / 6.3 GB	0.0 B	4
4	172.36.70.76:39323	Active	0	0.0 B / 6.3 GB	0.0 B	4

In questo esempio, il numero di attività Spark (58100) è molto maggiore delle 16 attività che gli esecutori possono elaborare contemporaneamente (4 esecutori × 4 core).

Se osservi questi sintomi, prendi in considerazione la possibilità di ridimensionare il cluster. È possibile scalare la capacità del cluster utilizzando le seguenti opzioni:

- Abilita AWS Glue Auto Scaling: [Auto Scaling](#) è disponibile per AWS Glue i processi di estrazione, trasformazione e caricamento (ETL) e di streaming AWS Glue nella versione 3.0 o successiva. AWS Glue aggiunge e rimuove automaticamente i worker dal cluster in base al numero di partizioni in ogni fase o alla velocità con cui vengono generati i microbatch durante l'esecuzione del lavoro.

Se osservate una situazione in cui il numero di lavoratori non aumenta anche se l'Auto Scaling è abilitato, prendete in considerazione l'aggiunta manuale dei lavoratori. Tuttavia, tieni presente che il ridimensionamento manuale per una fase potrebbe far sì che molti lavoratori rimangano inattivi nelle fasi successive, con costi maggiori senza alcun miglioramento delle prestazioni.

Dopo aver abilitato Auto Scaling, puoi vedere il numero di executor nelle metriche degli executor. CloudWatch Utilizza le seguenti metriche per monitorare la richiesta di esecutori nelle applicazioni Spark:

- `glue.driver.ExecutorAllocationManager.executors.numberAllExecutors`
- `glue.driver.ExecutorAllocationManager.executors.numberMaxNeededExecutors`

Per ulteriori informazioni sui parametri, consulta [Monitoraggio AWS Glue tramite i CloudWatch parametri Amazon](#).

- Scalabilità orizzontale: aumenta il numero di AWS Glue lavoratori: puoi aumentare manualmente il numero di AWS Glue lavoratori. Aggiungi lavoratori solo finché non osservi i lavoratori inattivi. A quel punto, l'aggiunta di più lavoratori aumenterà i costi senza migliorare i risultati. Per ulteriori informazioni, consulta [Parallelizzare le attività](#).
- Scalabilità verticale: utilizza un tipo di worker più grande: puoi modificare manualmente il tipo di istanza dei tuoi AWS Glue lavoratori per utilizzare lavoratori con più core, memoria e storage. I tipi di worker più grandi consentono di scalare verticalmente ed eseguire lavori intensivi di integrazione dei dati, come trasformazioni di dati che richiedono molta memoria, aggregazioni asimmetriche e controlli di rilevamento delle entità che coinvolgono petabyte di dati.

La scalabilità è utile anche nei casi in cui il driver Spark necessiti di una maggiore capacità, ad esempio perché il piano di richieste di lavoro è piuttosto ampio. Per ulteriori informazioni sui tipi di lavoratori e sulle prestazioni, consulta il post del AWS Big Data Blog [Scale your AWS Glue for Apache Spark](#) jobs with new large worker types and. G.4X G.8X

L'utilizzo di lavoratori più grandi può anche ridurre il numero totale di lavoratori necessari, il che aumenta le prestazioni riducendo la dispersione nelle operazioni intensive come la partecipazione.

Usa la versione più recente AWS Glue version

Ti consigliamo di utilizzare la versione più recente AWS Glue . In ogni versione sono presenti diverse ottimizzazioni e aggiornamenti che potrebbero migliorare automaticamente le prestazioni lavorative. Ad esempio, la AWS Glue versione 4.0 offre le seguenti nuove funzionalità:

- Il nuovo runtime ottimizzato di Apache Spark 3.3.0 — AWS Glue 4.0 si basa sul runtime Apache Spark 3.3.0 e offre miglioramenti prestazionali comparabili a quelli di Spark open source. Il runtime Spark 3.3.0 si basa su molte delle innovazioni di Spark 2.x.

- Connettore Amazon Redshift avanzato: AWS Glue 4.0 e versioni successive forniscono l'integrazione di Amazon Redshift per Apache Spark. L'integrazione si basa su un connettore open source esistente e lo migliora in termini di prestazioni e sicurezza. L'integrazione aiuta le applicazioni a prestazioni fino a 10 volte più veloci. Per ulteriori informazioni, consulta il post di blog sull'[integrazione di Amazon Redshift con Apache Spark](#).
- SIMD-based esecuzione per letture vettoriali con dati CSV e JSON: la AWS Glue versione 3.0 e le versioni successive aggiungono lettori ottimizzati che possono velocizzare notevolmente le prestazioni lavorative complessive rispetto ai lettori basati su righe. Per ulteriori informazioni sui dati CSV, consulta [Ottimizzazione](#) delle prestazioni di lettura con il lettore CSV SIMD vettorializzato. Per ulteriori informazioni sui dati JSON, consulta [Utilizzo del lettore JSON SIMD vettorializzato con il formato colonnare](#) Apache Arrow.

Ogni AWS Glue versione includerà aggiornamenti di questo tipo, tra i tanti, tra cui connettori, aggiornamenti di driver e librerie. Per ulteriori informazioni, consulta [AWS Glue Versioni](#) e [migrazione dei AWS Glue lavori alla AWS Glue versione 4.0](#).

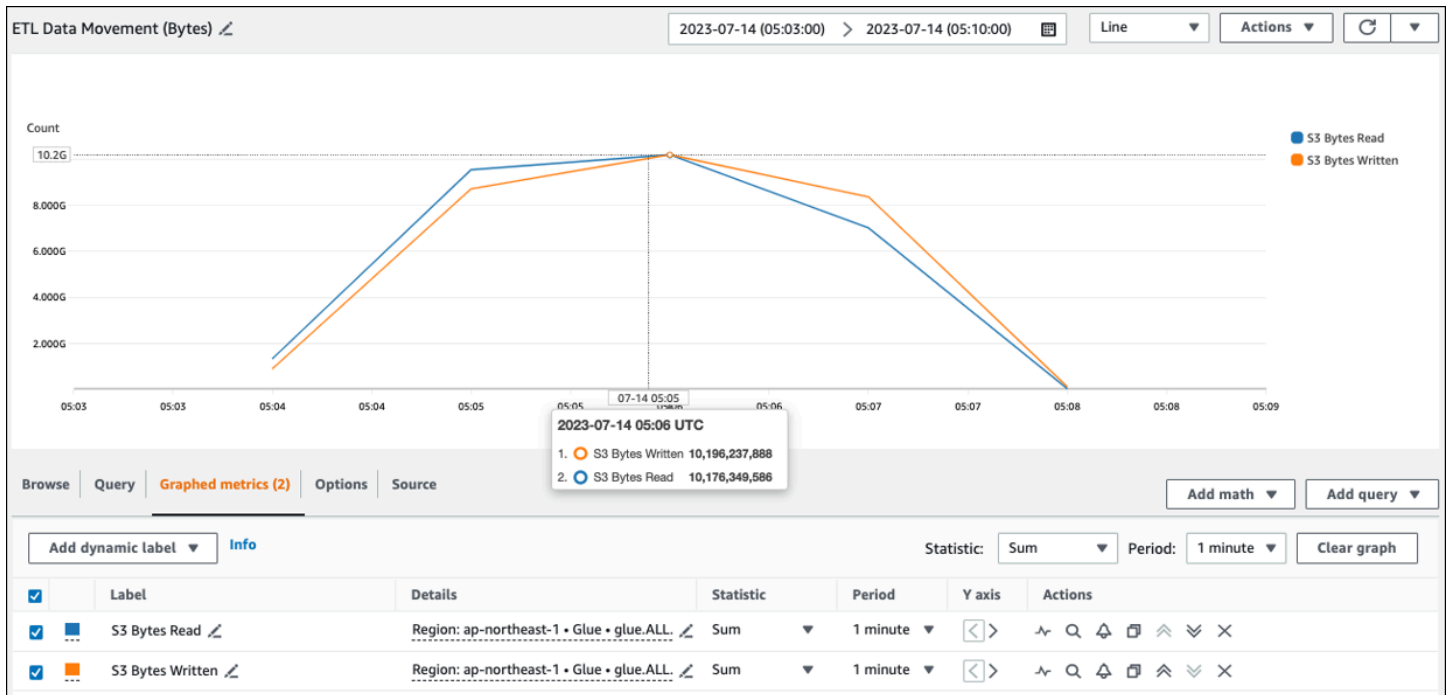
Ridurre la quantità di scansione dei dati

Per iniziare, valuta la possibilità di caricare solo i dati di cui hai bisogno. Puoi migliorare le prestazioni semplicemente riducendo la quantità di dati caricati nel cluster Spark per ogni fonte di dati. Per valutare se questo approccio è appropriato, utilizza le seguenti metriche.

[Puoi controllare i byte letti da Amazon S3 CloudWatch nelle metriche e maggiori dettagli nell'interfaccia utente di Spark, come descritto nella sezione Interfaccia utente Spark.](#)

CloudWatch metriche

Puoi vedere la dimensione approssimativa di lettura da Amazon S3 [in ETL Data Movement](#) (Bytes). Questa metrica mostra il numero di byte letti da Amazon S3 da tutti gli esecutori rispetto al rapporto precedente. Puoi utilizzarlo per monitorare il movimento dei dati ETL da Amazon S3 e confrontare le velocità di lettura con quelle di ingestione da fonti di dati esterne.



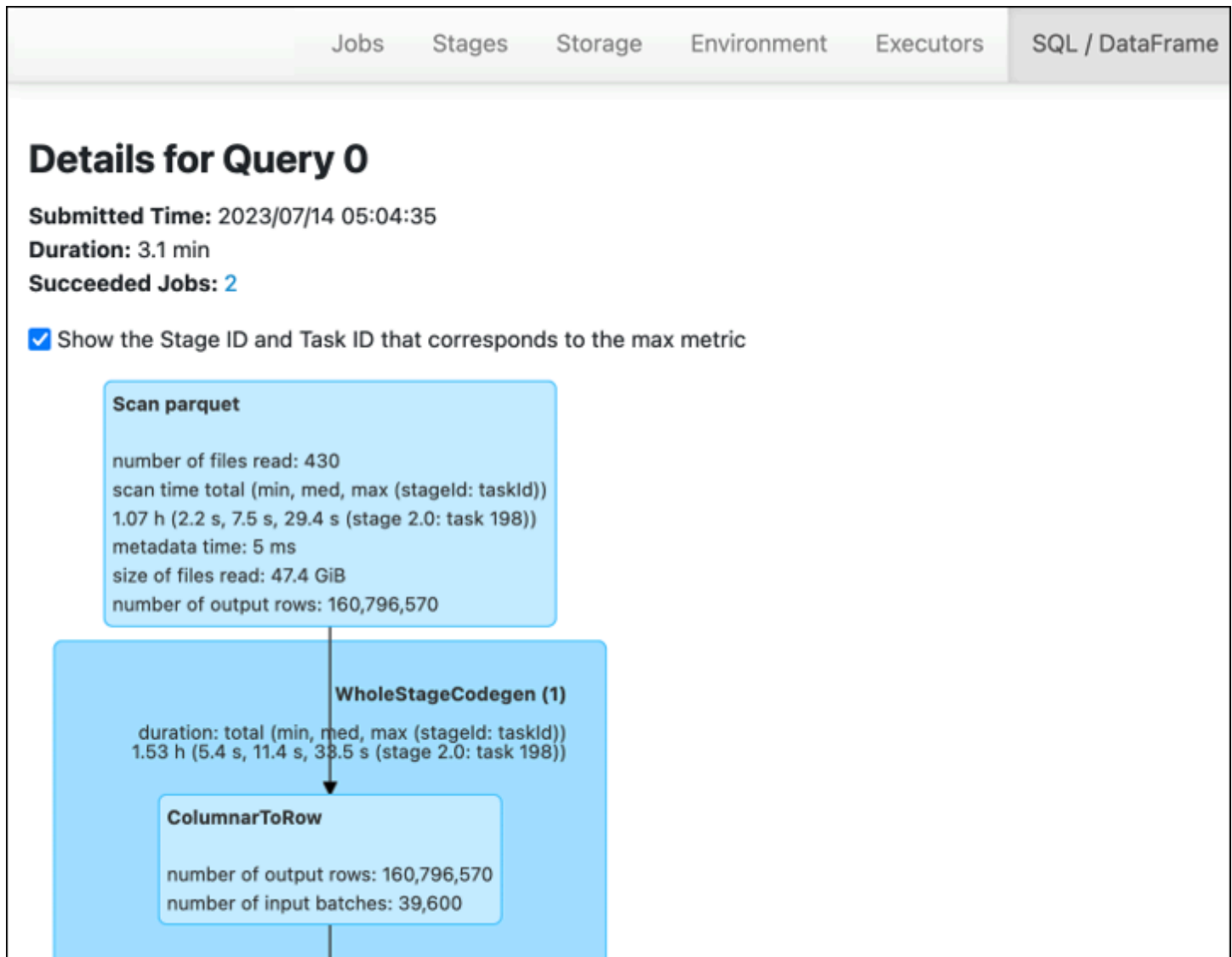
Se osservi un data point S3 Bytes Read più grande del previsto, prendi in considerazione le seguenti soluzioni.

Interfaccia utente Spark

Nella scheda Stage dell'interfaccia utente AWS Glue di Spark, puoi vedere le dimensioni di input e output. Nell'esempio seguente, lo stage 2 legge 47,4 GiB di input e 47,7 GiB di output, mentre lo stage 5 legge 61,2 MiB di input e 56,6 MiB di output.

Stages for All Jobs						
Completed Stages: 6						
Completed Stages (6)						
Page: 1		1 Pages. Jump to 1 . Sho				
Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output
5	parquet at NativeMethodAccessorImpl.java:0	2023/07/14 05:09:49	15 s	414/414	61.2 MiB	56.6 MiB
4	load at NativeMethodAccessorImpl.java:0	2023/07/14 05:09:47	0.6 s	1/1		
3	Listing leaf files and directories for 43 paths: s3://amazon-reviews-pds/parquet/product_category=Apparel, ... load at NativeMethodAccessorImpl.java:0	2023/07/14 05:09:46	1 s	43/43		
2	parquet at NativeMethodAccessorImpl.java:0	2023/07/14 05:04:36	3.1 min	414/414	47.4 GiB	47.7 GiB
1	load at NativeMethodAccessorImpl.java:0	2023/07/14 05:04:31	2 s	1/1		
0	Listing leaf files and directories for 43 paths: s3://amazon-reviews-pds/parquet/product_category=Apparel, ... load at NativeMethodAccessorImpl.java:0	2023/07/14 05:04:13	6 s	43/43		

Quando utilizzi lo Spark SQL o gli DataFrame approcci nel tuo AWS Glue lavoro, la scheda SQL/ mostra ulteriori statistiche su queste fasi. DataFrame In questo caso, la fase 2 mostra il numero di file letti: 430, la dimensione dei file letti: 47,4 GiB e il numero di righe di output: 160.796.570.



Se notate che esiste una differenza sostanziale nelle dimensioni tra i dati che state leggendo e quelli che state utilizzando, provate le seguenti soluzioni.

Simple Storage Service (Amazon S3)

Per ridurre la quantità di dati caricati nel lavoro durante la lettura da Amazon S3, considera le dimensioni, la compressione, il formato del file e il layout del file (partizioni) per il set di dati. AWS Glue I job for Spark vengono spesso utilizzati per l'ETL di dati grezzi, ma per un'elaborazione distribuita efficiente è necessario controllare le caratteristiche del formato di origine dei dati.

- Dimensioni del file: consigliamo di mantenere le dimensioni dei file di input e output entro un intervallo moderato (ad esempio, 128 MB). File troppo piccoli e file troppo grandi possono causare problemi.

Un numero elevato di file di piccole dimensioni causa i seguenti problemi:

- Carico di rete elevato su Amazon S3 a causa del I/O sovraccarico necessario per effettuare richieste (come `List`, `Get`, o `Head`) per molti oggetti (rispetto a pochi oggetti che memorizzano la stessa quantità di dati).
- Carico di elaborazione elevato I/O sul driver Spark, che genererà molte partizioni e attività e porterà a un parallelismo eccessivo.

D'altra parte, se il tipo di file non è divisibile (come gzip) e i file sono troppo grandi, l'applicazione Spark deve attendere il completamento di una singola operazione di lettura dell'intero file.

[Per ridurre l'eccessivo parallelismo che si verifica quando viene creata un'attività di Apache Spark per ogni file di piccole dimensioni, utilizzate il raggruppamento di file per `DynamicFrames`](#)

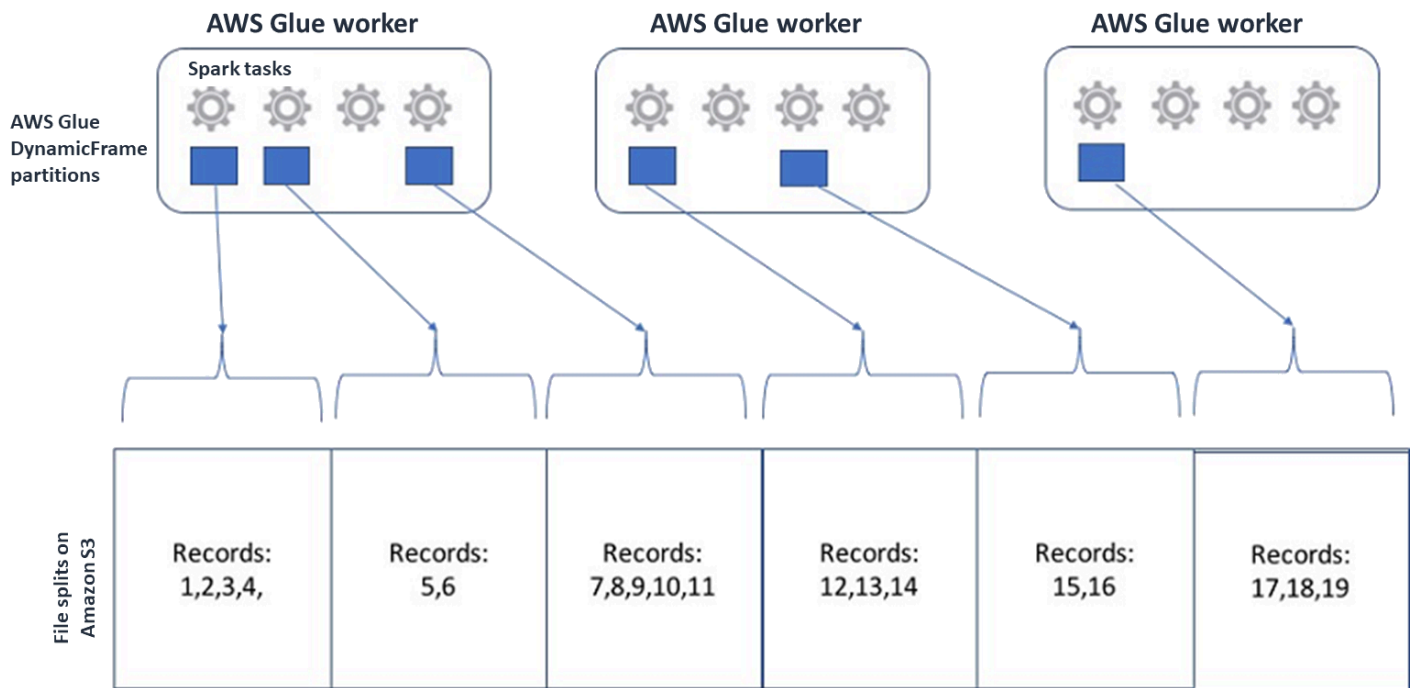
Questo approccio riduce le possibilità di un'eccezione OOM dal driver Spark. Per configurare il raggruppamento dei file, imposta i parametri `groupFiles` and `groupSize`. Il seguente esempio di codice utilizza l' AWS Glue `DynamicFrame` API in uno script ETL con questi parametri.

```
dyf = glueContext.create_dynamic_frame_from_options("s3",
    {'paths': ["s3://input-s3-path/"],
    'recurse': True,
    'groupFiles': 'inPartition',
    'groupSize': '1048576'},
    format="json")
```

- Compressione: se i tuoi oggetti S3 misurano centinaia di megabyte, valuta la possibilità di comprimerli. Esistono vari formati di compressione, che possono essere generalmente classificati in due tipi:
 - I formati di compressione non divisibili come gzip richiedono che l'intero file venga decompresso da un operatore.
 - I formati di compressione divisibili, come bzip2 o LZO (indicizzati), consentono la decompressione parziale di un file, che può essere parallelizzata.

Per Spark (e altri motori di elaborazione distribuita comuni), suddividerai il file di dati sorgente in blocchi che il tuo motore può elaborare in parallelo. Queste unità vengono spesso chiamate split. Dopo che i dati sono in un formato divisibile, i AWS Glue lettori ottimizzati possono recuperare le

suddivisioni da un oggetto S3 offrendo all'GetObjectAPI la Range possibilità di recuperare solo blocchi specifici. Considerate il diagramma seguente per vedere come funzionerebbe in pratica.



I dati compressi possono velocizzare notevolmente l'applicazione, purché i file abbiano una dimensione ottimale o siano divisibili. Le dimensioni dei dati più piccole riducono i dati scansionati da Amazon S3 e il traffico di rete da Amazon S3 al cluster Spark. D'altra parte, è necessaria più CPU per comprimere e decomprimere i dati. La quantità di calcolo richiesta varia in base al rapporto di compressione dell'algoritmo di compressione. Considerate questo compromesso quando scegliete il formato di compressione divisibile.

Note

Sebbene i file gzip non siano generalmente divisibili, puoi comprimere singoli blocchi di parquet con gzip e quei blocchi possono essere parallelizzati.

- Formato di file: utilizza un formato colonnare. [Apache Parquet e Apache ORC sono i formati di dati colonnari più diffusi](#). Parquet e ORC archiviano i dati in modo efficiente utilizzando la compressione, la codifica e la compressione basate su colonne in base al tipo di dati. [Per ulteriori informazioni sulle codifiche Parquet, consulta Definizioni di codifica Parquet](#). I file Parquet sono anche divisibili.

I formati colonnari raggruppano i valori per colonna e li memorizzano in blocchi. Quando usi i formati colonnari, puoi saltare i blocchi di dati che corrispondono a colonne che non intendi utilizzare. Le applicazioni Spark possono recuperare solo le colonne necessarie. In genere, rapporti di compressione migliori o il salto di blocchi di dati implicano la lettura di un numero inferiore di byte da Amazon S3, con conseguente miglioramento delle prestazioni. Entrambi i formati supportano anche i seguenti approcci pushdown per ridurre: I/O

- Projection pushdown: il projection pushdown è una tecnica per recuperare solo le colonne specificate nell'applicazione. È possibile specificare le colonne nell'applicazione Spark, come mostrato negli esempi seguenti:
 - DataFrame esempio: `df.select("star_rating")`
 - Esempio di SQL Spark: `spark.sql("select star_rating from <table>")`
- Predicate pushdown: il predicate pushdown è una tecnica per l'elaborazione efficiente di clausole e clausole. WHERE GROUP BY Entrambi i formati hanno blocchi di dati che rappresentano i valori delle colonne. Ogni blocco contiene le statistiche relative al blocco, ad esempio i valori massimi e minimi. Spark può utilizzare queste statistiche per determinare se il blocco deve essere letto o ignorato a seconda del valore del filtro utilizzato nell'applicazione. Per utilizzare questa funzione, aggiungi altri filtri nelle condizioni, come mostrato negli esempi seguenti:
 - DataFrame esempio: `df.select("star_rating").filter("star_rating < 2")`
 - Esempio di SQL Spark: `spark.sql("select * from <table> where star_rating < 2")`
- Layout dei file: archiviando i dati S3 su oggetti in percorsi diversi in base a come verranno utilizzati i dati, puoi recuperare in modo efficiente i dati pertinenti. Per ulteriori informazioni, consulta [Organizzazione degli oggetti utilizzando i prefissi nella documentazione](#) di Amazon S3. AWS Glue supporta la memorizzazione di chiavi e valori nei prefissi Amazon S3 nel formato `key=value`, partizionando i dati in base al percorso Amazon S3. Partizionando i dati, puoi limitare la quantità di dati scansionati da ciascuna applicazione di analisi downstream, migliorando le prestazioni e riducendo i costi. Per ulteriori informazioni, consulta [Gestione delle partizioni](#) per l'output ETL in AWS Glue

Il partizionamento divide la tabella in diverse parti e mantiene i dati correlati in file raggruppati in base ai valori delle colonne come anno, mese e giorno, come illustrato nell'esempio seguente.

```
# Partitioning by /YYYY/MM/DD
s3://<YourBucket>/year=2023/month=03/day=31/0000.gz
s3://<YourBucket>/year=2023/month=03/day=01/0000.gz
```

```
s3://<YourBucket>/year=2023/month=03/day=02/0000.gz
s3://<YourBucket>/year=2023/month=03/day=03/0000.gz
...
```

È possibile definire le partizioni per il set di dati modellandolo con una tabella in. AWS Glue Data Catalog. È quindi possibile limitare la quantità di dati scansionati utilizzando la rimozione delle partizioni come segue:

- Per AWS Glue DynamicFrame, set `push_down_predicate` (`ocatalogPartitionPredicate`).

```
dyf = Glue_context.create_dynamic_frame.from_catalog(
    database=src_database_name,
    table_name=src_table_name,
    push_down_predicate = "year='2023' and month = '03'",
)
```

- Per Spark DataFrame, imposta un percorso fisso per eliminare le partizioni.

```
df = spark.read.format("json").load("s3://<YourBucket>/year=2023/month=03/*/*.gz")
```

- Per Spark SQL, puoi impostare la clausola `where` per eliminare le partizioni dal Data Catalog.

```
df = spark.sql("SELECT * FROM <Table> WHERE year= '2023' and month = '03'")
```

- Per partizionare per data quando si scrivono i dati con AWS Glue, si imposta [partitionKeys](#) DynamicFrame in [o partitionBy\(\)](#) con le informazioni sulla data DataFrame nelle colonne come segue.

- DynamicFrame

```
glue_context.write_dynamic_frame_from_options(
    frame= dyf, connection_type='s3', format='parquet'
    connection_options= {
        'partitionKeys': ["year", "month", "day"],
        'path': 's3://<YourBucket>/<Prefix>/'
    }
)
```

- DataFrame

```
df.write.mode('append')\
```

```
.partitionBy('year','month','day')\
.parquet('s3://<YourBucket>/<Prefix>/')
```

Ciò può migliorare le prestazioni degli utenti dei dati di output.

Se non avete accesso per modificare la pipeline che crea il set di dati di input, il partizionamento non è un'opzione. Invece, puoi escludere percorsi S3 non necessari utilizzando modelli a glob. Imposta le [esclusioni durante](#) la lettura. DynamicFrame Ad esempio, il codice seguente esclude i giorni nei mesi da 01 a 09, nell'anno 2023.

```
dyf = glueContext.create_dynamic_frame.from_catalog(
    database=db,
    table_name=table,
    additional_options = { "exclusions":["\\\"**year=2023/month=0[1-9]**\\\""] },
    transformation_ctx='dyf'
)
```

Puoi anche impostare esclusioni nelle proprietà della tabella nel Catalogo dati:

- Chiave: `exclusions`
- Valore: `[\"**year=2023/month=0[1-9]**\"]`
- Troppe partizioni Amazon S3: evita di partizionare i dati Amazon S3 su colonne che contengono un'ampia gamma di valori, ad esempio una colonna ID con migliaia di valori. Ciò può aumentare notevolmente il numero di partizioni nel bucket, poiché il numero di partizioni possibili è il prodotto di tutti i campi in base ai quali hai partizionato. Troppe partizioni potrebbero causare quanto segue:
 - Maggiore latenza per il recupero dei metadati delle partizioni dal Data Catalog
 - Aumento del numero di file di piccole dimensioni, che richiede più richieste API Amazon S3 (`ListGet`, `e`) `Head`

Ad esempio, quando si imposta un tipo di data in `partitionBy` o `partitionKeys`, il partizionamento a livello di data `yyyy/mm/dd` è utile per molti casi d'uso. Tuttavia, `yyyy/mm/dd/<ID>` potrebbe generare così tante partizioni da influire negativamente sulle prestazioni complessive.

D'altra parte, alcuni casi d'uso, come le applicazioni di elaborazione in tempo reale, richiedono molte partizioni come `yyyy/mm/dd/hh`. Se il tuo caso d'uso richiede partizioni sostanziali, prendi in considerazione l'utilizzo di [indici di AWS Glue partizione](#) per ridurre la latenza per il recupero dei metadati delle partizioni dal Data Catalog.

Database e JDBC

Per ridurre la scansione dei dati durante il recupero di informazioni da un database, è possibile specificare un where predicato (o una clausola) in una query SQL. I database che non forniscono un'interfaccia SQL forniranno il proprio meccanismo per l'interrogazione o il filtraggio.

Quando si utilizzano connessioni Java Database Connectivity (JDBC), fornite una query di selezione con la where clausola per i seguenti parametri:

- [Per DynamicFrame, utilizzate l'opzione SampleQuery.](#) Durante l'utilizzo di `create_dynamic_frame.from_catalog`, configurate l'additional_options argomento come segue.

```
query = "SELECT * FROM <TableName> where id = 'XX' AND"
datasource0 = glueContext.create_dynamic_frame.from_catalog(
    database = db,
    table_name = table,
    additional_options={
        "sampleQuery": query,
        "hashexpression": key,
        "hashpartitions": 10,
        "enablePartitioningForSampleQuery": True
    },
    transformation_ctx = "datasource0"
)
```

Quando si utilizza `create_dynamic_frame.from_options`, configura l'connection_options argomento come segue.

```
query = "SELECT * FROM <TableName> where id = 'XX' AND"
datasource0 = glueContext.create_dynamic_frame.from_options(
    connection_type = connection,
    connection_options={
        "url": url,
        "user": user,
        "password": password,
        "dbtable": table,
        "sampleQuery": query,
        "hashexpression": key,
        "hashpartitions": 10,
        "enablePartitioningForSampleQuery": True
    }
)
```

```
}  
)
```

- Per DataFrame, usa l'opzione di [interrogazione](#).

```
query = "SELECT * FROM <TableName> where id = 'XX'"  
jdbcDF = spark.read \  
.format('jdbc') \  
.option('url', url) \  
.option('user', user) \  
.option('password', pwd) \  
.option('query', query) \  
.load()
```

- Per Amazon Redshift, usa la AWS Glue versione 4.0 o successiva per sfruttare il supporto pushdown nel connettore [Amazon](#) Redshift Spark.

```
dyf = glueContext.create_dynamic_frame.from_catalog(  
    database = "redshift-dc-database-name",  
    table_name = "redshift-table-name",  
    redshift_tmp_dir = args["temp-s3-dir"],  
    additional_options = {"aws_iam_role": "arn:aws:iam::role-account-id:role/rs-role-  
name"}  
)
```

- Per altri database, consulta la documentazione relativa a quel database.

AWS Glue options

- Per evitare una scansione completa per tutte le esecuzioni continue dei lavori ed elaborare solo i dati che non erano presenti durante l'ultima esecuzione del processo, abilita i [segnalibri dei lavori](#).
- Per limitare la quantità di dati di input da elaborare, abilita l'[esecuzione limitata](#) con i segnalibri dei lavori. Questo aiuta a ridurre la quantità di dati scansionati per ogni esecuzione di lavoro.

Parallelizza le attività

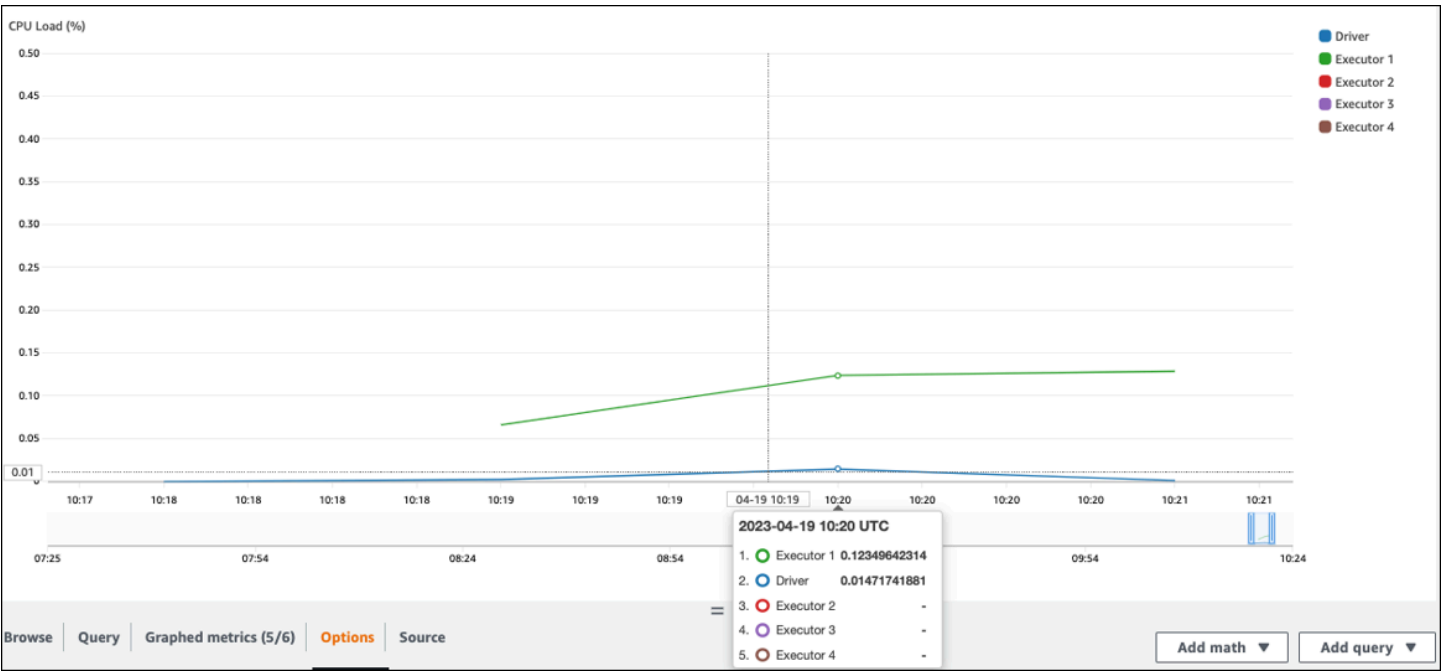
Per ottimizzare le prestazioni, è importante parallelizzare le attività per i caricamenti e le trasformazioni dei dati. Come abbiamo discusso [in Argomenti chiave di Apache Spark](#), il numero di partizioni resilienti di set di dati distribuiti (RDD) è importante, perché determina il grado di

parallelismo. Ogni attività creata da Spark corrisponde a una partizione RDD su base 1:1. Per ottenere le migliori prestazioni, è necessario capire come viene determinato il numero di partizioni RDD e come tale numero viene ottimizzato.

Se il parallelismo non è sufficiente, i seguenti sintomi verranno registrati nelle [CloudWatchmetriche](#) e nell'interfaccia utente di Spark.

CloudWatch metriche

Controlla il carico della CPU e l'utilizzo della memoria. Se alcuni esecutori non eseguono l'elaborazione durante una fase del lavoro, è opportuno migliorare il parallelismo. In questo caso, durante il periodo di tempo visualizzato, l'Executor 1 stava eseguendo un'operazione, ma gli esecutori rimanenti (2, 3 e 4) no. Si può dedurre che a quegli esecutori non sono stati assegnati compiti dal driver Spark.

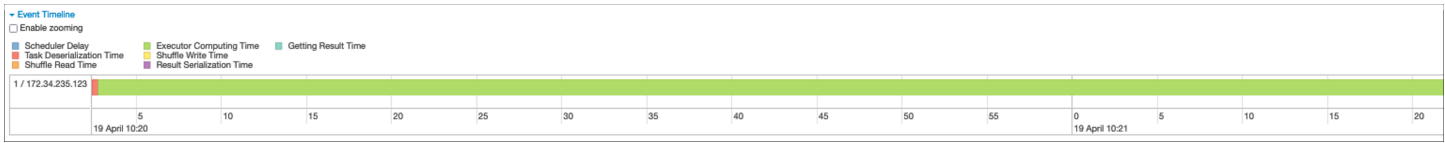


Interfaccia utente Spark

Nella scheda Stage dell'interfaccia utente Spark, puoi vedere il numero di attività in una fase. In questo caso, Spark ha eseguito una sola operazione.

Tasks (1)														
Index	ID	Attempt	Status	Locality Level	Executor ID	Host	Launch Time	Duration	Task Deserialization Time	GC Time	Result Serialization Time	Input Size / Records	Write Time	Shuffle Write Size / Records
0	1	0	SUCCESS	ANY	1	172.34.235.123	2023/04/19 10:20:02	1.3 min	0.3 s	0.4 s	1 ms	2.0 GB / 7135819	12 ms	59.0 B / 1

Inoltre, la sequenza temporale dell'evento mostra l'Executor 1 che elabora un'operazione. Ciò significa che il lavoro in questa fase è stato eseguito interamente su un esecutore, mentre gli altri erano inattivi.



Se osservi questi sintomi, prova le seguenti soluzioni per ogni fonte di dati.

Parallelizza il caricamento dei dati da Amazon S3

Per parallelizzare i caricamenti di dati da Amazon S3, controlla innanzitutto il numero predefinito di partizioni. Puoi quindi determinare manualmente il numero di partizioni di destinazione, ma assicurati di evitare di avere troppe partizioni.

Determina il numero predefinito di partizioni

Per Amazon S3, il numero iniziale di partizioni Spark RDD (ognuna delle quali corrisponde a un'attività Spark) è determinato dalle caratteristiche del set di dati Amazon S3 (ad esempio, formato, compressione e dimensione). Quando crei uno AWS Glue DynamicFrame o uno Spark DataFrame da oggetti CSV archiviati in Amazon S3, il numero iniziale di partizioni RDD `NumPartitions()` può essere calcolato approssimativamente come segue:

- Dimensione dell'oggetto ≤ 64 MB: `NumPartitions = Number of Objects`
- Dimensione dell'oggetto > 64 MB: `NumPartitions = Total Object Size / 64 MB`
- Indivisibile (gzip): `NumPartitions = Number of Objects`

Come discusso nella sezione [Ridurre la quantità di scansione dei dati](#), Spark divide oggetti S3 di grandi dimensioni in suddivisioni che possono essere elaborate in parallelo. Quando l'oggetto è più grande della dimensione suddivisa, Spark lo divide e crea una partizione RDD (e un'attività) per ogni divisione. La dimensione suddivisa di Spark si basa sul formato dei dati e sull'ambiente di runtime, ma questa è un'approssimazione iniziale ragionevole. Alcuni oggetti vengono compressi utilizzando formati di compressione non divisibili come gzip, quindi Spark non può dividerli.

Il `NumPartitions` valore può variare in base al formato dei dati, alla compressione, alla AWS Glue versione, al numero di worker e alla configurazione di AWS Glue Spark.

Ad esempio, quando carichi un singolo `csv.gz` oggetto da 10 GB usando Spark DataFrame, il driver Spark creerà solo una partizione RDD (`NumPartitions=1`) perché gzip non è divisibile. Ciò comporta un carico pesante su un particolare esecutore Spark e nessun compito viene assegnato agli esecutori rimanenti, come descritto nella figura seguente.

Controlla il numero effettivo di attività (`NumPartitions`) per lo stage nella scheda [Spark Web UI](#) Stage oppure esegui il codice per verificare il `df.rdd.getNumPartitions()` parallelismo.

Quando trovi un file gzip da 10 GB, esamina se il sistema che lo genera è in grado di generarlo in un formato divisibile. Se questa non è un'opzione, potrebbe essere necessario [scalare la capacità del cluster](#) per elaborare il file. Per eseguire le trasformazioni in modo efficiente sui dati caricati, è necessario ribilanciare l'RDD tra i worker del cluster utilizzando la ripartizione.

Determina manualmente il numero di partizioni di destinazione

A seconda delle proprietà dei dati e dell'implementazione di alcune funzionalità da parte di Spark, potresti ritrovarti con un `NumPartitions` valore basso anche se il lavoro sottostante può ancora essere parallelizzato. Se `NumPartitions` è troppo piccolo, esegui `df.repartition(N)` per aumentare il numero di partizioni in modo che l'elaborazione possa essere distribuita su più esecutori Spark.

In questo caso, l'esecuzione `df.repartition(100)` passerà `NumPartitions` da 1 a 100, creando 100 partizioni di dati, ognuna con un'attività che può essere assegnata agli altri esecutori.

L'operazione `repartition(N)` divide equamente tutti i dati (10 GB/ 100 partizioni = 100 MB/partition), evitando la distorsione dei dati su determinate partizioni.

Note

Quando viene eseguita un'operazione di shuffle come quella eseguita, il numero di partizioni `join` viene aumentato o diminuito dinamicamente in base al valore di `o.spark.sql.shuffle.partitions` `spark.default.parallelism`. Ciò facilita uno scambio di dati più efficiente tra gli esecutori Spark. [Per ulteriori informazioni, consulta la documentazione di Spark.](#)

Il vostro obiettivo nel determinare il numero di partizioni previsto è quello di massimizzare l'uso dei lavoratori assegnati. AWS Glue Il numero di AWS Glue lavoratori e il numero di attività Spark

sono correlati dal numero di vCPU. Spark supporta un'attività per ogni core vCPU. Nella AWS Glue versione 3.0 o successiva, puoi calcolare un numero target di partizioni utilizzando la seguente formula.

```
# Calculate NumPartitions by WorkerType
numExecutors = (NumberOfWorkers - 1)
numSlotsPerExecutor =
  4 if WorkerType is G.1X
  8 if WorkerType is G.2X
 16 if WorkerType is G.4X
 32 if WorkerType is G.8X
NumPartitions = numSlotsPerExecutor * numExecutors

# Example: Glue 4.0 / G.1X / 10 Workers
numExecutors = ( 10 - 1 ) = 9 # 1 Worker reserved on Spark Driver
numSlotsPerExecutor = 4 # G.1X has 4 vCpu core ( Glue 3.0 or later )
NumPartitions = 9 * 4 = 36
```

In questo esempio, ogni G.1X worker fornisce quattro core vCPU a un executor Spark (). `spark.executor.cores = 4` Spark supporta un'attività per ogni vCPU Core, G.1X quindi gli esecutori Spark possono eseguire quattro attività contemporaneamente (). `numSlotPerExecutor` Questo numero di partizioni sfrutta appieno il cluster se le attività richiedono lo stesso periodo di tempo. Tuttavia, alcune attività richiederanno più tempo di altre, creando core inattivi. In tal caso, valuta la possibilità di moltiplicare `numPartitions` per 2 o 3 per suddividere e pianificare in modo efficiente le attività più difficili.

Troppe partizioni

Un numero eccessivo di partizioni crea un numero eccessivo di attività. Ciò causa un carico pesante sul driver Spark a causa del sovraccarico legato all'elaborazione distribuita, come le attività di gestione e lo scambio di dati tra gli esecutori Spark.

Se il numero di partizioni del lavoro è notevolmente superiore al numero di partizioni previsto, valuta la possibilità di ridurre il numero di partizioni. È possibile ridurre le partizioni utilizzando le seguenti opzioni:

- Se le dimensioni dei file sono molto piccole, usa AWS Glue [GroupFiles](#). È possibile ridurre l'eccessivo parallelismo derivante dall'avvio di un'attività di Apache Spark per elaborare ogni file.
- Si usa per unire le `coalesce(N)` partizioni. Si tratta di un processo a basso costo. Quando si riduce il numero di partizioni, `coalesce(N)` è preferibile rispetto a `primarepartition(N)`,

perché `repartition(N)` esegue lo shuffle per distribuire equamente la quantità di record in ogni partizione. Ciò aumenta i costi e il sovraccarico di gestione.

- Usa Spark 3.x Adaptive Query Execution. Come discusso nella sezione [Argomenti chiave di Apache Spark](#), Adaptive Query Execution fornisce una funzione per unire automaticamente il numero di partizioni. È possibile utilizzare questo approccio quando non è possibile conoscere il numero di partizioni fino a quando non si esegue l'esecuzione.

Parallelizza il caricamento dei dati da JDBC

Il numero di partizioni Spark RDD è determinato dalla configurazione. Nota che per impostazione predefinita viene eseguita solo una singola attività per scansione un intero set di dati di origine tramite una query. `SELECT`

AWS Glue DynamicFrames Sia Spark che Spark DataFrames supportano il caricamento parallelo di dati JDBC su più attività. Questa operazione viene eseguita utilizzando i `where` predicati per suddividere una query in più query. `SELECT` Per parallelizzare le letture da JDBC, configura le seguenti opzioni:

- Per AWS Glue DynamicFrame, set (`o and`, `hashfield` `hashexpression`) `hashpartition` Per ulteriori informazioni, consulta [Leggere da tabelle JDBC in parallelo](#).

```
connection_mysql8_options = {
    "url": "jdbc:mysql://XXXXXXXXXX.XXXXXXX.us-east-1.rds.amazonaws.com:3306/test",
    "dbtable": "medicare_tb",
    "user": "test",
    "password": "XXXXXXXXXX",
    "hashexpression": "id",
    "hashpartitions": "10"
}
datasource0 = glueContext.create_dynamic_frame.from_options(
    'mysql',
    connection_options=connection_mysql8_options,
    transformation_ctx= "datasource0"
)
```

- Per Spark DataFrame, set, `numPartitions` `partitionColumn`, `lowerBound` e `upperBound` Per ulteriori informazioni, consulta [JDBC To Other Databases](#).

```
df = spark.read \
```

```
.format("jdbc") \
.option("url", "jdbc:mysql://XXXXXXXXXX.XXXXXXX.us-east-1.rds.amazonaws.com:3306/
test") \
.option("dbtable", "medicare_tb") \
.option("user", "test") \
.option("password", "XXXXXXXXXX") \
.option("partitionColumn", "id") \
.option("numPartitions", "10") \
.option("lowerBound", "0") \
.option("upperBound", "1141455") \
.load()

df.write.format("json").save("s3://bucket_name/Tests/sparkjdbc/with_parallel/")
```

Parallelizza il caricamento dei dati da DynamoDB quando usi il connettore ETL

Il numero di partizioni Spark RDD è determinato dal parametro. `dynamodb.splits` Per parallelizzare le letture da Amazon DynamoDB, configura le seguenti opzioni:

- Aumenta il valore di `dynamodb.splits`
- Ottimizza il parametro seguendo la formula spiegata in [Tipi di connessione e opzioni per ETL in AWS Glue for Spark](#).

Parallelizza il caricamento dei dati da Kinesis Data Streams

Il numero di partizioni Spark RDD è determinato dal numero di shard nel flusso di dati Amazon Kinesis Data Streams di origine. Se hai solo pochi shard nel tuo flusso di dati, ci saranno solo alcune attività Spark. Ciò può comportare un basso parallelismo nei processi a valle. Per parallelizzare le letture da Kinesis Data Streams, configura le seguenti opzioni:

- Aumenta il numero di shard per ottenere un maggiore parallelismo durante il caricamento dei dati da Kinesis Data Streams.
- Se la logica del microbatch è abbastanza complessa, valuta la possibilità di ripartizionare i dati all'inizio del batch, dopo aver eliminato le colonne non necessarie.

Per ulteriori informazioni, consulta [Best practice per ottimizzare costi e prestazioni per lo streaming di lavori ETL](#). AWS Glue

Parallelizza le attività dopo il caricamento dei dati

Per parallelizzare le attività dopo il caricamento dei dati, aumenta il numero di partizioni RDD utilizzando le seguenti opzioni:

- Ripartiziona i dati per generare un numero maggiore di partizioni, soprattutto subito dopo il caricamento iniziale se non è possibile parallelizzare il carico stesso.

Chiama su `repartition()` `DynamicFrame` o `DataFrame`, specificando il numero di partizioni. Una buona regola empirica è due o tre volte il numero di core disponibili.

Tuttavia, quando si scrive una tabella partizionata, ciò può portare a un'esplosione di file (ogni partizione può potenzialmente generare un file in ogni partizione della tabella). Per evitare ciò, puoi ripartizionare il file per colonna. `DataFrame` Questo utilizza le colonne di partizione della tabella in modo che i dati siano organizzati prima della scrittura. È possibile specificare un numero maggiore di partizioni senza inserire file di piccole dimensioni nelle partizioni della tabella. Tuttavia, fate attenzione a evitare la distorsione dei dati, in cui alcuni valori di partizione finiscono con la maggior parte dei dati e ritardano il completamento dell'operazione.

- In caso di mescolamenti, aumentate il valore. `spark.sql.shuffle.partitions` Questo può anche aiutare a risolvere eventuali problemi di memoria durante lo shuffling.

Quando hai più di 2.001 partizioni shuffle, Spark utilizza un formato di memoria compressa. Se hai un numero vicino a quello, potresti voler impostare il `spark.sql.shuffle.partitions` valore oltre tale limite per ottenere una rappresentazione più efficiente.

Ottimizza gli shuffles

Alcune operazioni, come `join()` e, richiedono che Spark esegua uno `groupByKey()` shuffle. Lo shuffle è il meccanismo di Spark per ridistribuire i dati in modo che siano raggruppati in modo diverso tra le partizioni RDD. Lo shuffling può aiutare a rimediare ai problemi di prestazioni. Tuttavia, poiché lo shuffling in genere implica la copia dei dati tra gli esecutori Spark, lo shuffle è un'operazione complessa e costosa. Ad esempio, lo shuffle genera i seguenti costi:

- Disco: I/O
 - Genera un gran numero di file intermedi su disco.
- Rete I/O:
 - Richiede molte connessioni di rete (Numero di connessioni = `Mapper` × `Reducer`).

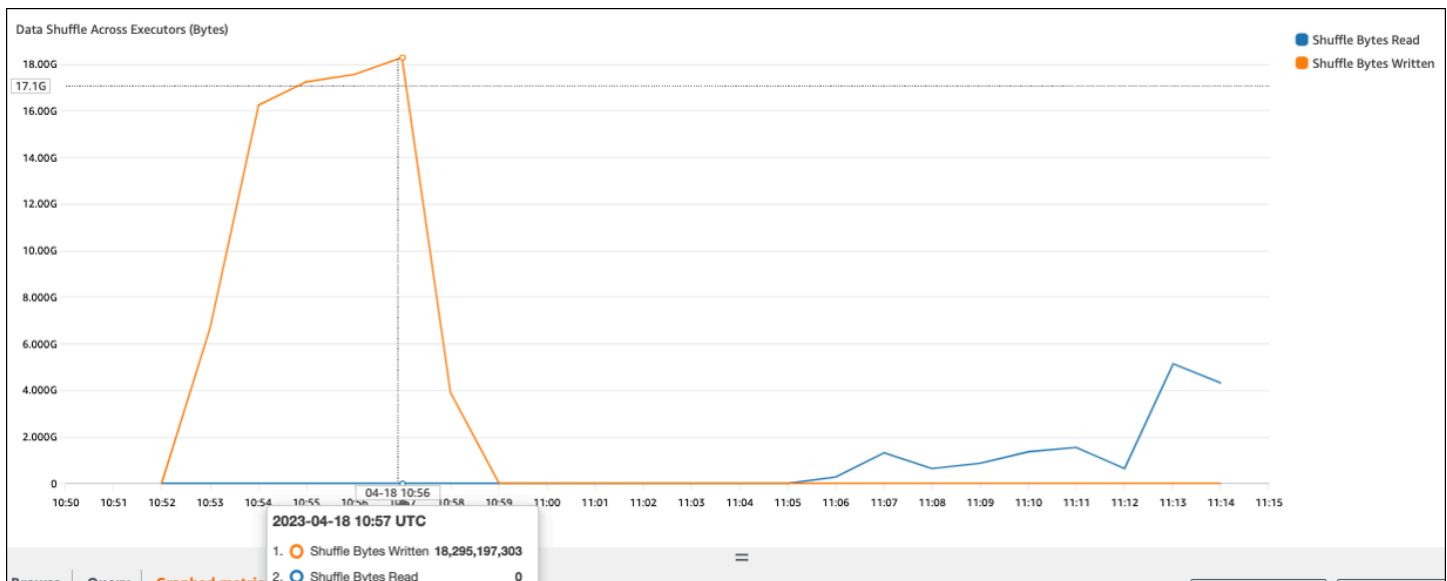
- Poiché i record vengono aggregati in nuove partizioni RDD che potrebbero essere ospitate su un altro esecutore Spark, una parte sostanziale del set di dati potrebbe spostarsi tra gli esecutori Spark sulla rete.
- Carico della CPU e della memoria:
 - Ordina i valori e unisce i set di dati. Queste operazioni sono pianificate sull'esecutore, con un carico pesante sull'esecutore.

Shuffle è uno dei fattori più importanti del peggioramento delle prestazioni dell'applicazione Spark. Durante la memorizzazione dei dati intermedi, può esaurire lo spazio sul disco locale dell'esecutore, causando il fallimento del job Spark.

Puoi valutare le tue prestazioni di shuffle nelle metriche e nell' CloudWatch interfaccia utente di Spark.

CloudWatch metriche

Se il valore Shuffle Bytes Written è elevato rispetto a Shuffle Bytes Read, il tuo job Spark potrebbe utilizzare operazioni di shuffle come `o. join()` `groupByKey()`



Interfaccia utente Spark

Nella scheda Stage dell'interfaccia utente Spark, puoi controllare i valori Shuffle Read Size/Records. Puoi vederlo anche nella scheda Executors.

Nella schermata seguente, ogni esecutore ne scambia circa 18. 6GB/4020000registra con il processo shuffle, per una dimensione totale di lettura casuale di circa 75 GB).

La colonna Shuffle Spill (Disk) mostra che una grande quantità di dati trasferisce memoria su disco, il che potrebbe causare un disco pieno o un problema di prestazioni.

- Aggregated Metrics by Executor				
Executor ID ▲	Address	Shuffle Read Size / Records	Shuffle Spill (Memory)	Shuffle Spill (Disk)
1	172.35.205.23:46731	18.6 GB / 40210300	98.1 GB	16.8 GB
2	172.35.195.173:46185	18.7 GB / 40246767	117.2 GB	17.3 GB
3	172.36.135.106:35913	18.6 GB / 40253921	101.6 GB	16.6 GB
4	172.34.131.223:46879	18.6 GB / 40190741	99.5 GB	16.4 GB

Se osservate questi sintomi e la fase richiede troppo tempo rispetto agli obiettivi prestazionali prefissati, oppure fallisce con Out Of Memory No space left on device errori, prendete in considerazione le seguenti soluzioni.

Ottimizza l'unione

L'`join()` operazione, che unisce le tabelle, è l'operazione di shuffle più comunemente utilizzata, ma è spesso un ostacolo alle prestazioni. Poiché l'unione è un'operazione costosa, si consiglia di non utilizzarla a meno che non sia essenziale per i requisiti aziendali. Double-check che state facendo un uso efficiente della vostra pipeline di dati ponendovi le seguenti domande:

- Stai ricalcolando un join che viene eseguito anche in altri lavori che puoi riutilizzare?
- Vi state unendo per risolvere chiavi esterne relative a valori che non vengono utilizzati dai consumatori del vostro output?

Dopo aver verificato che le operazioni di join sono essenziali per i requisiti aziendali, consulta le seguenti opzioni per ottimizzare l'iscrizione in modo da soddisfare i requisiti.

Usa il pushdown prima di unirti

Filtra le righe e le colonne non necessarie DataFrame prima di eseguire un join. Ciò presenta i seguenti vantaggi:

- Riduce la quantità di dati trasferiti durante lo shuffle
- Riduce la quantità di elaborazione nell'esecutore Spark

- Riduce la quantità di dati da scansionare

```
# Default
df_joined = df1.join(df2, ["product_id"])

# Use Pushdown
df1_select =
  df1.select("product_id", "product_title", "star_rating").filter(col("star_rating")>=4.0)
df2_select = df2.select("product_id", "category_id")
df_joined = df1_select.join(df2_select, ["product_id"])
```

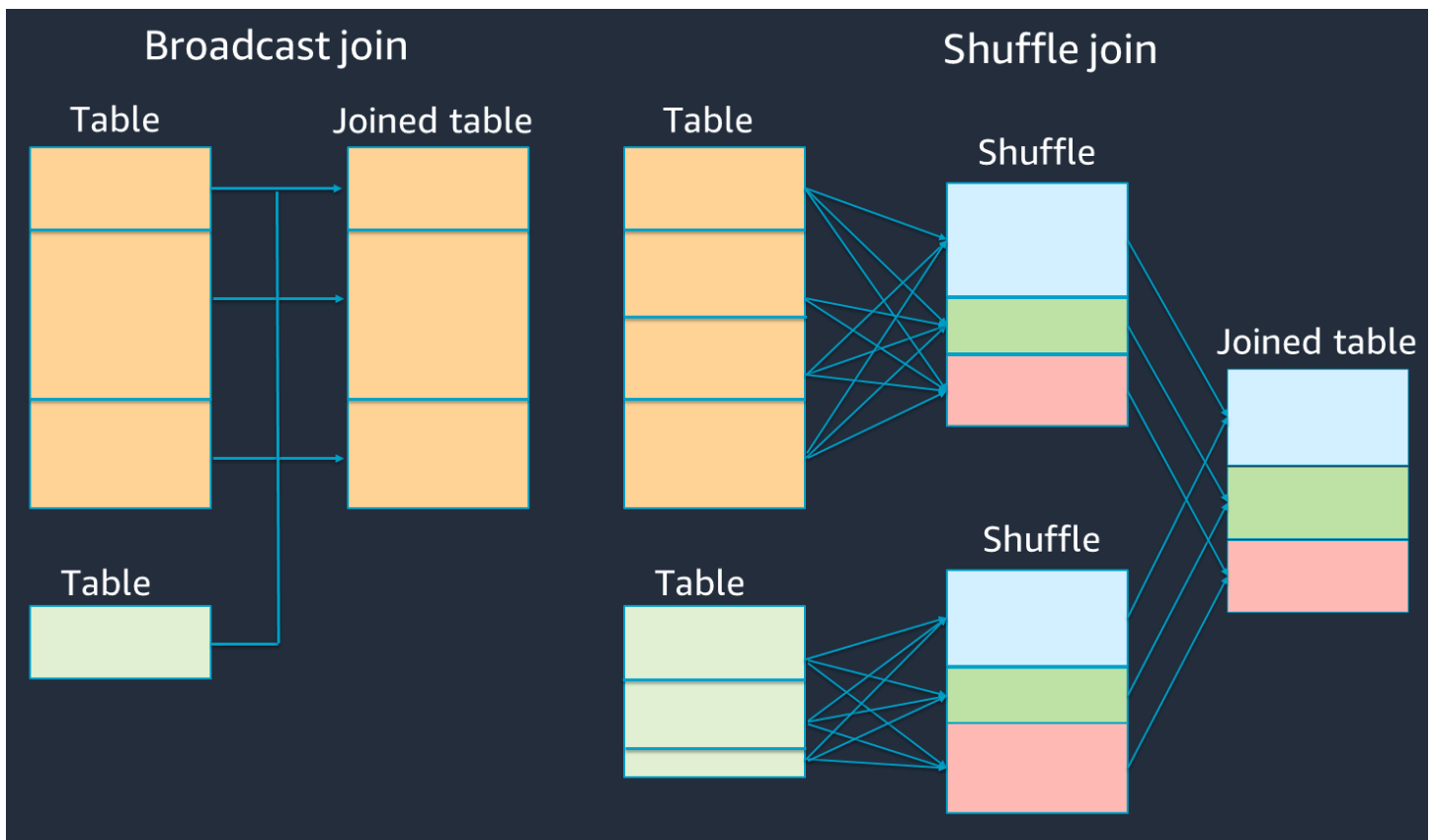
Usa DataFrame Join

Prova a utilizzare un'[API Spark di alto livello](#) come SparkSQL e Datasets anziché l'API RDD o join. DataFrame DynamicFrame Puoi eseguire la conversione in DataFrame con una DynamicFrame chiamata di metodo come. `dyf.toDF()` Come discusso nella sezione [Argomenti chiave di Apache Spark](#), queste operazioni di unione sfruttano internamente l'ottimizzazione delle query mediante l'ottimizzatore Catalyst.

Mescola e trasmetti hash, join e suggerimenti

Spark supporta due tipi di join: shuffle join e broadcast hash join. Un broadcast hash join non richiede lo shuffling e può richiedere meno elaborazione rispetto a uno shuffle join. Tuttavia, è applicabile solo quando si unisce un tavolo piccolo a uno grande. Quando ti unisci a una tabella che può stare nella memoria di un singolo esecutore Spark, prendi in considerazione l'utilizzo di un broadcast hash join.

Il diagramma seguente mostra la struttura e le fasi di alto livello di un broadcast hash join e di uno shuffle join.



I dettagli di ogni join sono i seguenti:

- Shuffle join:
 - Lo shuffle hash join unisce due tabelle senza ordinamento e distribuisce il join tra le due tabelle. È adatto per unire piccole tabelle che possono essere archiviate nella memoria dell'esecutore Spark.
 - Lo sort-merge join distribuisce le due tabelle da unire tramite chiave e le ordina prima di unirle. È adatto per unire tavoli di grandi dimensioni.
- Broadcast hash join:
 - Un broadcast hash join invia l'RDD o la tabella più piccola a ciascuno dei nodi di lavoro. Quindi esegue una combinazione sul lato della mappa con ciascuna partizione del RDD o della tabella più grande.

È adatto per i join quando uno dei tuoi RDD o delle tabelle può stare in memoria o può essere adattato alla memoria. È utile eseguire un hash join per la trasmissione quando possibile, perché non richiede un shuffle. Puoi usare un hint di iscrizione per richiedere un join alla trasmissione da Spark come segue.

```
# DataFrame
from pySpark.sql.functions import broadcast
df_joined= df_big.join(broadcast(df_small), right_df[key] == left_df[key],
    how='inner')

-- SparkSQL
SELECT /*+ BROADCAST(t1) / FROM t1 INNER JOIN t2 ON t1.key = t2.key;
```

[Per maggiori informazioni sui suggerimenti per partecipare, vedi Join hints.](#)

Nella AWS Glue versione 3.0 e successive, puoi sfruttare automaticamente i broadcast hash join abilitando [Adaptive Query](#) Execution e parametri aggiuntivi. Adaptive Query Execution converte un sort-merge join in un broadcast hash join quando le statistiche di runtime di entrambi i lati del join sono inferiori alla soglia di adaptive broadcast hash join.

Nella AWS Glue 3.0, è possibile abilitare Adaptive Query Execution impostando.
`spark.sql.adaptive.enabled=true` L'esecuzione adattiva delle query è abilitata per impostazione predefinita in AWS Glue 4.0.

Puoi impostare parametri aggiuntivi relativi agli shuffle e agli hash join di trasmissione:

- `spark.sql.adaptive.localShuffleReader.enabled`
- `spark.sql.adaptive.autoBroadcastJoinThreshold`

Per ulteriori informazioni sui parametri correlati, vedere [Conversione di sort-merge](#) join in broadcast join.

Nella AWS Glue versione 3.0 e versioni successive, puoi usare altri suggerimenti di join for shuffle per ottimizzare il tuo comportamento.

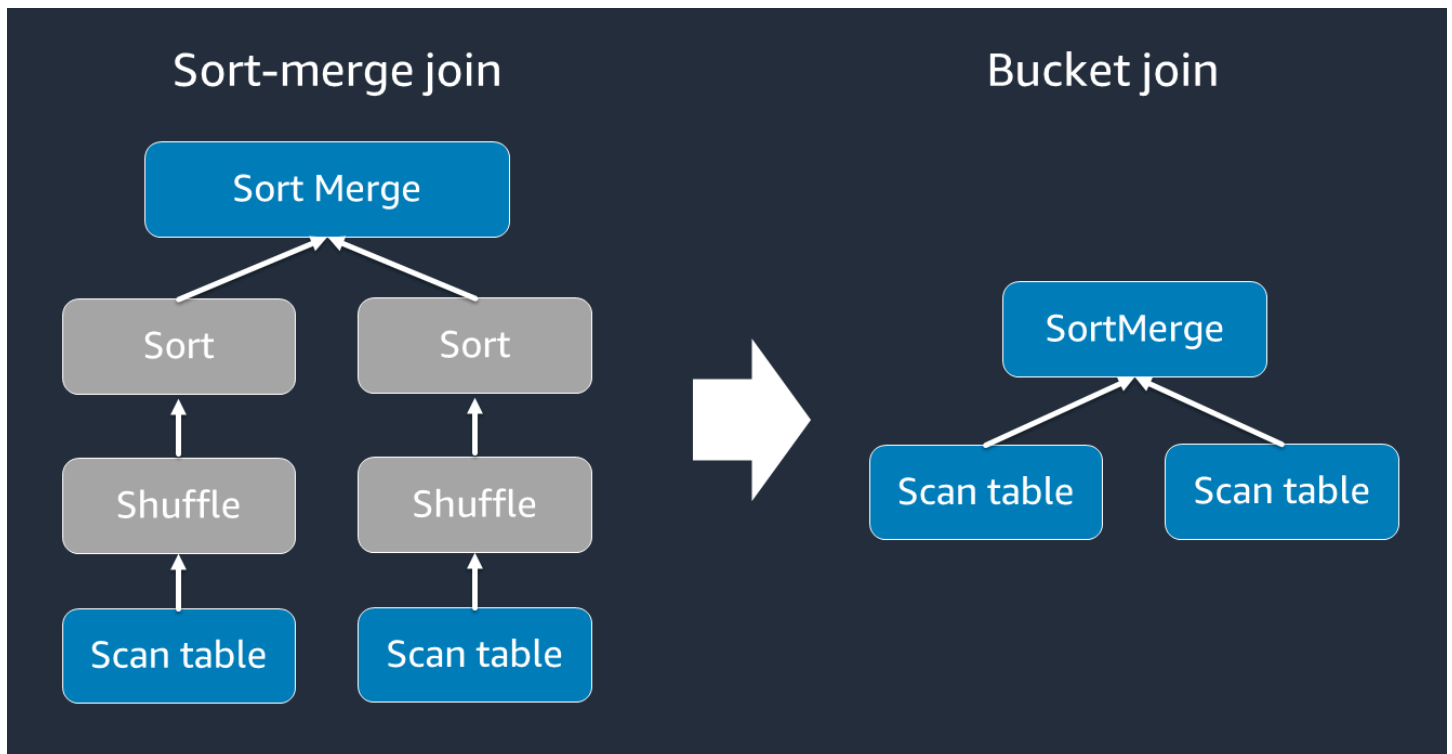
```
-- Join Hints for shuffle sort merge join
SELECT /*+ SHUFFLE_MERGE(t1) / FROM t1 INNER JOIN t2 ON t1.key = t2.key;
SELECT /*+ MERGEJOIN(t2) / FROM t1 INNER JOIN t2 ON t1.key = t2.key;
SELECT /*+ MERGE(t1) / FROM t1 INNER JOIN t2 ON t1.key = t2.key;

-- Join Hints for shuffle hash join
SELECT /*+ SHUFFLE_HASH(t1) / FROM t1 INNER JOIN t2 ON t1.key = t2.key;
```

```
-- Join Hints for shuffle-and-replicate nested loop join
SELECT /*+ SHUFFLE_REPLICATE_NL(t1) */ FROM t1 INNER JOIN t2 ON t1.key = t2.key;
```

Usa il bucketing

Il sort-merge join richiede due fasi, shuffle e sort, quindi merge. Queste due fasi possono sovraccaricare l'esecutore Spark e causare problemi di OOM e prestazioni quando alcuni esecutori si uniscono e altri si ordinano contemporaneamente. [In questi casi, potrebbe essere possibile unire in modo efficiente utilizzando il bucketing.](#) Bucketing preordinerà e mescolerà i dati immessi nelle chiavi di join, quindi scriverà i dati ordinati in una tabella intermedia. Il costo delle fasi di mescolamento e ordinamento può essere ridotto quando si uniscono tabelle di grandi dimensioni definendo in anticipo le tabelle intermedie ordinate.



Le tabelle bucketed sono utili per quanto segue:

- I dati vengono uniti frequentemente sulla stessa chiave, ad esempio `account_id`
- Caricamento di tabelle cumulative giornaliere, ad esempio tabelle base e delta, che potrebbero essere inserite in una colonna comune

È possibile creare una tabella a blocchi utilizzando il codice seguente.

```
df.write.bucketBy(50, "account_id").sortBy("age").saveAsTable("bucketed_table")
```

Ripartizione delle DataFrames chiavi di unione prima dell'unione

Per ripartizionare le due chiavi DataFrames sulle chiavi di join prima del join, utilizzate le seguenti istruzioni.

```
df1_repartitioned = df1.repartition(N,"join_key")  
df2_repartitioned = df2.repartition(N,"join_key")  
df_joined = df1_repartitioned.join(df2_repartitioned,"product_id")
```

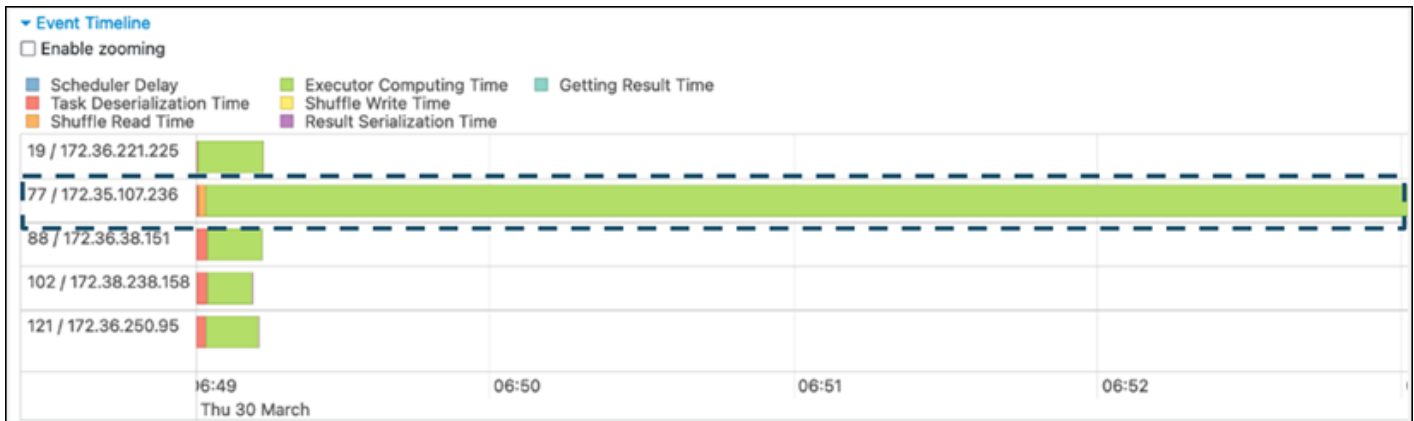
Questo partiziona due RDD (ancora separati) sulla chiave di join prima di iniziare l'unione. Se i due RDD sono partizionati sulla stessa chiave con lo stesso codice di partizionamento, è molto probabile che i record RDD che intendi unire vengano collocati sullo stesso worker prima di procedere all'unione. Ciò potrebbe migliorare le prestazioni riducendo l'attività di rete e la distorsione dei dati durante l'unione.

Supera la distorsione dei dati

La distorsione dei dati è una delle cause più comuni di difficoltà per i lavori Spark. Si verifica quando i dati non sono distribuiti uniformemente tra le partizioni RDD. Ciò fa sì che le attività per quella partizione richiedano molto più tempo rispetto ad altre, ritardando il tempo di elaborazione complessivo dell'applicazione.

Per identificare la distorsione dei dati, valuta le seguenti metriche nell'interfaccia utente di Spark:

- Nella scheda Stage dell'interfaccia utente Spark, esamina la pagina Cronologia degli eventi. Puoi vedere una distribuzione non uniforme delle attività nella schermata seguente. Le attività distribuite in modo non uniforme o che richiedono troppo tempo per essere eseguite possono indicare una distorsione dei dati.



- Un'altra pagina importante è Summary Metrics, che mostra le statistiche per le attività di Spark. La schermata seguente mostra le metriche con percentili per Duration, GC Time, Spill (memoria), Spill (disco) e così via.

Summary Metrics for 5 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	9 s	10 s	11 s	13 s	6.4 min
GC Time	0.0 ms	0.2 s	0.3 s	0.4 s	1 s
Spill (memory)	0.0 B	0.0 B	0.0 B	0.0 B	16.7 GiB
Spill (disk)	0.0 B	0.0 B	0.0 B	0.0 B	10.2 GiB
Output Size / Records	8.3 MiB / 12651	9.4 MiB / 21462	36.1 MiB / 63860	92.9 MiB / 258057	10.1 GiB / 20370130
Shuffle Read Size / Records	9.8 MiB / 12651	11.7 MiB / 21462	43.4 MiB / 63860	122.6 MiB / 258057	11.8 GiB / 20370130

Quando le attività sono distribuite uniformemente, vedrai numeri simili in tutti i percentili. In caso di distorsione dei dati, in ogni percentile verranno visualizzati valori molto distorti. Nell'esempio, la durata dell'attività è inferiore a 13 secondi in Min, 25° percentile, Mediano e 75° percentile. Sebbene l'attività Max abbia elaborato 100 volte più dati rispetto al 75° percentile, la sua durata di 6,4 minuti è circa 30 volte maggiore. Significa che almeno un'attività (o fino al 25 per cento delle attività) ha richiesto molto più tempo rispetto al resto delle attività.

Se i dati sono distorti, prova quanto segue:

- Se usi la AWS Glue versione 3.0, abilita Adaptive Query Execution impostando.
`spark.sql.adaptive.enabled=true` Adaptive Query Execution è abilitato per impostazione predefinita nella AWS Glue versione 4.0.

È inoltre possibile utilizzare Adaptive Query Execution per la distorsione dei dati introdotta dai join impostando i seguenti parametri correlati:

- `spark.sql.adaptive.skewJoin.skewedPartitionFactor`
- `spark.sql.adaptive.skewJoin.skewedPartitionThresholdInBytes`

- `spark.sql.adaptive.advisoryPartitionSizeInBytes=128m` (128 mebibytes or larger should be good)
- `spark.sql.adaptive.coalescePartitions.enabled=true` (when you want to coalesce partitions)

Per ulteriori informazioni, consulta la documentazione di [Apache Spark](#).

- Usa chiavi con un ampio intervallo di valori per le chiavi di unione. In uno shuffle join, le partizioni vengono determinate per ogni valore hash di una chiave. Se la cardinalità di una chiave di join è troppo bassa, è più probabile che la funzione hash faccia un cattivo lavoro di distribuzione dei dati tra le partizioni. Pertanto, se la tua applicazione e la tua logica aziendale la supportano, prendi in considerazione l'utilizzo di una chiave di cardinalità più elevata o di una chiave composita.

```
# Use Single Primary Key
df_joined = df1_select.join(df2_select, ["primary_key"])

# Use Composite Key
df_joined = df1_select.join(df2_select, ["primary_key", "secondary_key"])
```

Usa la cache

Quando usi metodi ripetitivi DataFrames, evita ulteriori operazioni di shuffle o calcoli utilizzando `df.cache()` o `df.persist()` memorizzando nella cache i risultati del calcolo nella memoria e su disco di ogni esecutore Spark. [Spark supporta anche RDD persistenti su disco o la replica su più nodi \(livello di storage\)](#).

Ad esempio, puoi rendere persistente aggiungendo. DataFrames `df.persist()` Quando la cache non è più necessaria, è possibile utilizzarla `unpersist` per eliminare i dati memorizzati nella cache.

```
df = spark.read.parquet("s3://<Bucket>/parquet/product_category=Books/")
df_high_rate = df.filter(col("star_rating")>=4.0)
df_high_rate.persist()

df_joined1 = df_high_rate.join(<Table1>, ["key"])
df_joined2 = df_high_rate.join(<Table2>, ["key"])
df_joined3 = df_high_rate.join(<Table3>, ["key"])
...
df_high_rate.unpersist()
```

Rimuovi le azioni Spark non necessarie

Evita di eseguire azioni non necessarie come `countshow`, o. `collect`. Come discusso nella sezione [Argomenti chiave di Apache Spark, Spark](#) è pigro. Ogni RDD trasformato potrebbe essere ricalcolato ogni volta che si esegue un'azione su di esso. Quando si utilizzano molte azioni Spark, vengono richiamati accessi a più sorgenti, calcoli di attività ed esecuzioni casuali per ciascuna azione.

Se non hai bisogno `collect()` di altre azioni nel tuo ambiente commerciale, valuta la possibilità di rimuoverle.

Note

Evita il più possibile di usare Spark `collect()` in ambienti commerciali. L'`collect()` azione restituisce tutti i risultati di un calcolo nell'esecutore Spark al driver Spark, il che potrebbe far sì che il driver Spark restituisca un errore OOM. Per evitare un errore OOM, Spark imposta di `spark.driver.maxResultSize = 1GB` default, che limita la dimensione massima dei dati restituiti al driver Spark a 1 GB.

Riduci al minimo i costi di pianificazione

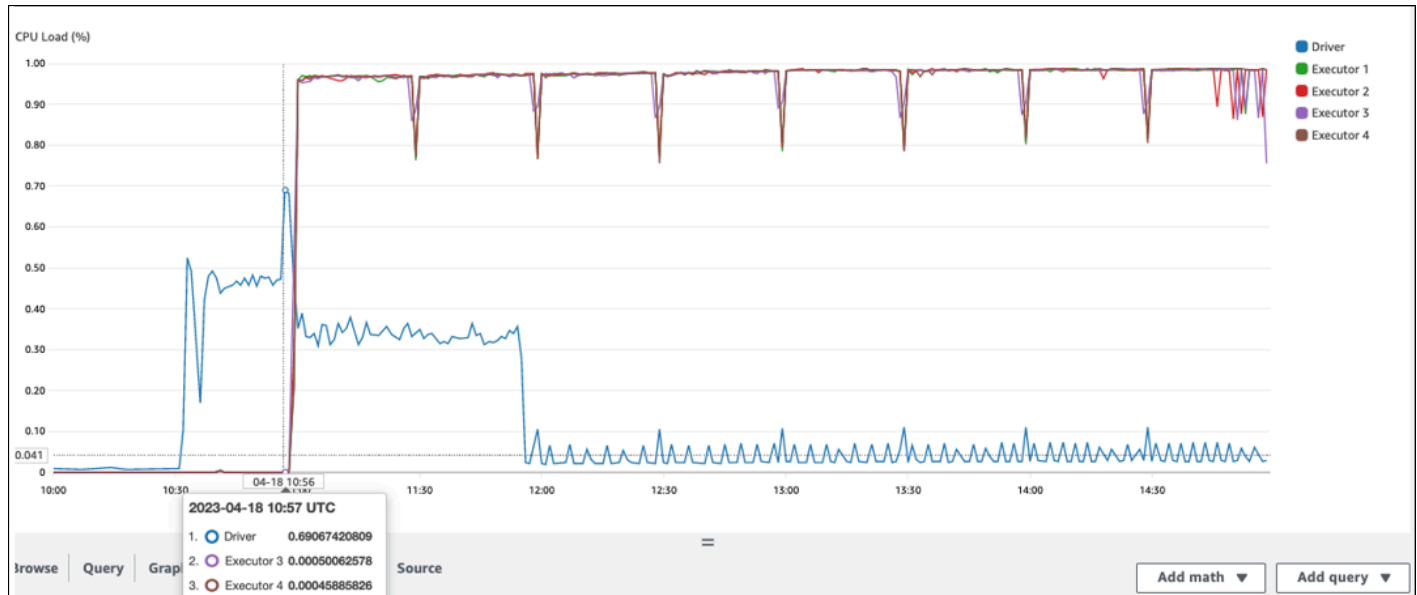
Come discusso [Argomenti chiave di Apache Spark, il driver Spark genera](#) il piano di esecuzione. In base a tale piano, le attività vengono assegnate all'esecutore Spark per l'elaborazione distribuita. Tuttavia, il driver Spark può diventare un collo di bottiglia se è presente un gran numero di file di piccole dimensioni o se AWS Glue Data Catalog contiene un gran numero di partizioni. Per identificare un sovraccarico di pianificazione elevato, valuta le seguenti metriche.

CloudWatch metriche

Controlla il carico della CPU e l'utilizzo della memoria per le seguenti situazioni:

- Il carico della CPU e l'utilizzo della memoria del driver Spark sono registrati come elevati. Normalmente, il driver Spark non elabora i dati, quindi il carico della CPU e l'utilizzo della memoria non subiscono picchi. Tuttavia, se l'origine dati di Amazon S3 contiene troppi file di piccole dimensioni, elencare tutti gli oggetti S3 e gestire un gran numero di attività potrebbe causare un utilizzo elevato delle risorse.
- C'è un lungo intervallo prima che l'elaborazione inizi in Spark Executor. Nella seguente schermata di esempio, il carico della CPU dell'esecutore Spark è troppo basso fino alle 10:57, anche se il

lavoro è iniziato alle 10:00. AWS Glue Ciò indica che il driver Spark potrebbe impiegare molto tempo per generare un piano di esecuzione. In questo esempio, recuperare il gran numero di partizioni nel Data Catalog ed elencare il gran numero di file di piccole dimensioni nel driver Spark richiede molto tempo.



Interfaccia utente Spark

Nella scheda Job dell'interfaccia utente Spark, puoi vedere l'ora di invio. Nell'esempio seguente, il driver Spark ha avviato job0 alle 10:56:46, anche se il lavoro è iniziato alle 10:00:00. AWS Glue

Completed Jobs (1)					
Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
0	count at DynamicFrame.scala:1414 count at DynamicFrame.scala:1414	2023/04/18 10:56:46	4.9 h	1/1	58100/58100

Puoi anche vedere l' Succeeded/Totalora delle attività (per tutte le fasi) nella scheda Job. In questo caso, il numero di attività viene registrato come58100. Come spiegato nella sezione Amazon S3 della pagina delle attività di [Parallelize, il numero di attività](#) corrisponde approssimativamente al numero di oggetti S3. Ciò significa che ci sono circa 58.100 oggetti in Amazon S3.

Per maggiori dettagli su questo lavoro e sulla tempistica, consulta la scheda Stage. Se riscontri un problema con il driver Spark, prendi in considerazione le seguenti soluzioni:

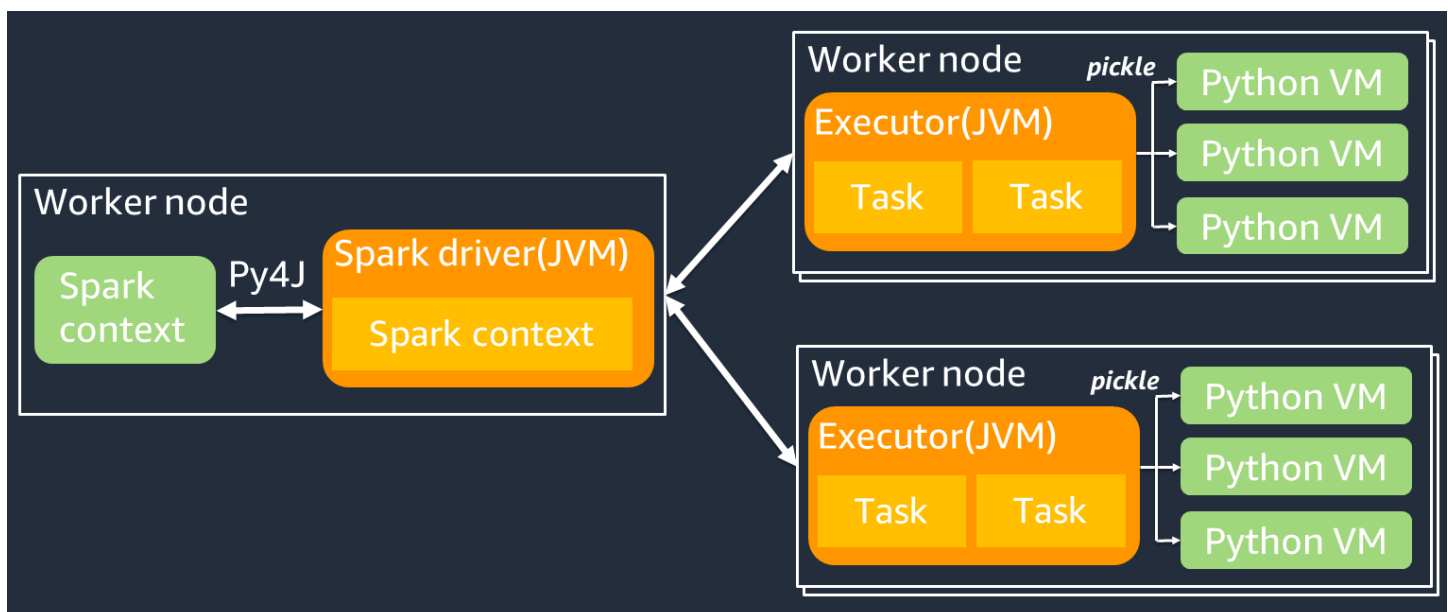
- [Quando Amazon S3 ha troppi file, prendi in considerazione le indicazioni sull'eccessivo parallelismo nella sezione Troppe partizioni della pagina delle attività di Parallelize.](#)

- [Quando Amazon S3 ha troppe partizioni, prendi in considerazione le indicazioni sul partizionamento eccessivo nella sezione Troppe partizioni Amazon S3 della pagina Riduci la quantità di scansione dei dati.](#) Abilita gli [indici di AWS Glue partizione](#) se ci sono molte partizioni per ridurre la latenza per il recupero dei metadati delle partizioni dal Data Catalog. [Per ulteriori informazioni, consulta Migliorare le prestazioni delle query utilizzando gli indici di partizione. AWS Glue](#)
- Quando JDBC ha troppe partizioni, riduci il valore. `hashpartition`
- Se DynamoDB ha troppe partizioni, riduci il valore. `dynamodb.splits`
- Quando i job di streaming hanno troppe partizioni, riduci il numero di shard.

Ottimizza le funzioni definite dall'utente

User-defined funzioni (UDF) e PySpark spesso riducono `RDD.map` in modo significativo le prestazioni. Ciò è dovuto al sovraccarico richiesto per rappresentare accuratamente il codice Python nell'implementazione Scala sottostante di Spark.

Il diagramma seguente mostra l'architettura dei job. PySpark Quando si utilizza PySpark, il driver Spark utilizza la libreria Py4J per richiamare i metodi Java da Python. Quando si chiamano Spark SQL o funzioni DataFrame integrate, c'è poca differenza di prestazioni tra Python e Scala perché le funzioni vengono eseguite sulla JVM di ogni esecutore utilizzando un piano di esecuzione ottimizzato.



Se si utilizza la propria logica Python, ad esempio `usingmap/ mapPartitions/ udf`, l'attività verrà eseguita in un ambiente di runtime Python. La gestione di due ambienti comporta costi generali. Inoltre, i dati in memoria devono essere trasformati per essere utilizzati dalle funzioni integrate dell'ambiente di runtime JVM. Pickle è un formato di serializzazione utilizzato di default per lo scambio tra i runtime JVM e Python. Tuttavia, il costo di questo costo di serializzazione e deserializzazione è molto elevato, quindi le UDF scritte in Java o Scala sono più veloci delle UDF in Python.

Per evitare il sovraccarico di serializzazione e deserializzazione, considerate quanto segue: PySpark

- Usa le funzioni SQL Spark integrate: valuta la possibilità di sostituire la tua funzione UDF o di mappa con Spark SQL o funzioni integrate. DataFrame Quando si esegue Spark SQL o funzioni DataFrame integrate, c'è poca differenza di prestazioni tra Python e Scala perché le attività vengono gestite sulla JVM di ogni esecutore.
- Implementa le UDF in Scala o Java: prendi in considerazione l'utilizzo di una UDF scritta in Java o Scala, perché vengono eseguite sulla JVM.
- Usa Apache Arrow-based UDF per carichi di lavoro vettoriali: prendi in considerazione l'utilizzo di UDF. Arrow-based Questa funzionalità è nota anche come UDF vettorializzato (Pandas UDF). [Apache Arrow](#) è un formato di dati in memoria indipendente dalla lingua che AWS Glue può essere utilizzato per trasferire in modo efficiente i dati tra processi JVM e Python. Questo è attualmente molto vantaggioso per gli utenti di Python che lavorano con Panda o dati. NumPy

Arrow è un formato colonnare (vettoriale). Il suo utilizzo non è automatico e potrebbe richiedere alcune piccole modifiche alla configurazione o al codice per trarne il massimo vantaggio e garantire la compatibilità. Per maggiori dettagli e limitazioni, consulta [Apache Arrow in PySpark](#).

L'esempio seguente confronta una UDF incrementale di base in Python standard, come UDF vettorializzata e in Spark SQL.

UDF standard in Python

Il tempo di esempio è 3,20 (sec).

Esempio di codice

```
# DataSet
df = spark.range(10000000).selectExpr("id AS a","id AS b")
```

```
# UDF Example
def plus(a,b):
    return a+b
spark.udf.register("plus",plus)

df.selectExpr("count(plus(a,b))").collect()
```

Piano di esecuzione

```
== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=false
+- HashAggregate(keys=[], functions=[count(pythonUDF0#124)])
+- Exchange SinglePartition, ENSURE_REQUIREMENTS, [id=#580]
+- HashAggregate(keys=[], functions=[partial_count(pythonUDF0#124)])
+- Project [pythonUDF0#124]
+- BatchEvalPython [plus(a#116L, b#117L)], [pythonUDF0#124]
+- Project [id#114L AS a#116L, id#114L AS b#117L]
+- Range (0, 100000000, step=1, splits=16)
```

UDF vettorizzata

Il tempo di esempio è 0,59 (sec).

L'UDF vettorizzata è 5 volte più veloce dell'esempio UDF precedente. Controllando `Physical Plan`, si può vedere `ArrowEvalPython`, che questa applicazione è vettorizzata da Apache Arrow. Per abilitare l'UDF vettorizzato, è necessario specificare nel codice.

```
spark.sql.execution.arrow.pyspark.enabled = true
```

Esempio di codice

```
# Vectorized UDF
from pyspark.sql.types import LongType
from pyspark.sql.functions import count, pandas_udf

# Enable Apache Arrow Support
spark.conf.set("spark.sql.execution.arrow.pyspark.enabled", "true")

# DataSet
df = spark.range(100000000).selectExpr("id AS a", "id AS b")

# Annotate pandas_udf to use Vectorized UDF
```

```
@pandas_udf(LongType())
def pandas_plus(a,b):
    return a+b
spark.udf.register("pandas_plus",pandas_plus)

df.selectExpr("count(pandas_plus(a,b))").collect()
```

Piano di esecuzione

```
== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=false
+- HashAggregate(keys=[], functions=[count(pythonUDF0#1082L)],
  output=[count(pandas_plus(a, b))#1080L])
+- Exchange SinglePartition, ENSURE_REQUIREMENTS, [id=#5985]
+- HashAggregate(keys=[], functions=[partial_count(pythonUDF0#1082L)],
  output=[count#1084L])
+- Project [pythonUDF0#1082L]
+- ArrowEvalPython [pandas_plus(a#1074L, b#1075L)], [pythonUDF0#1082L], 200
+- Project [id#1072L AS a#1074L, id#1072L AS b#1075L]
+- Range (0, 100000000, step=1, splits=16)
```

Spark SQL

Il tempo di esempio è 0,087 (sec).

Spark SQL è molto più veloce dell'UDF vettorializzato, perché le attività vengono eseguite sulla JVM di ogni esecutore senza un runtime Python. Se puoi sostituire l'UDF con una funzione integrata, ti consigliamo di farlo.

Esempio di codice

```
df.createOrReplaceTempView("test")
spark.sql("select count(a+b) from test").collect()
```

Usare i panda per i big data

Se conosci già i panda e vuoi usare Spark per i big data, puoi usare l'API [pandas](#) su Spark. AWS Glue 4.0 e versioni successive lo supportano. Per iniziare, puoi utilizzare il notebook ufficiale [Quickstart: Pandas API](#) su Spark. Per ulteriori informazioni, consulta la [documentazione relativa ad PySpark](#).

Risorse

- [AWS Glue](#)
- [Ottimizzazione delle prestazioni](#) (Spark SQL Guide)
- [AWS Glue Workshop sull'ottimizzazione](#)

Cronologia dei documenti

La tabella seguente descrive le modifiche significative apportate a questa guida. Per ricevere notifiche sugli aggiornamenti futuri, puoi abbonarti a un [feed RSS](#).

Modifica	Descrizione	Data
Pubblicazione iniziale	—	2 gennaio 2024

Le traduzioni sono generate tramite traduzione automatica. In caso di conflitto tra il contenuto di una traduzione e la versione originale in Inglese, quest'ultima prevarrà.