リソースプール管理に関するベストプラクティス

February, 2018

原文はこちら

MICRO FOCUS

目次

はじめに

本ドキュメントは Vertica の管理者を対象としており、定義されたワークロードとアドホックなワークロードを的確に処理するためのカスタムリソースプールの作成に関するいくつかの例を示しています。
Vertica 上で、同時に複数のクエリを実行する必要が出てくるでしょう。システム上で複数のクエリを実行する場合、クエリはリソースを共有します。したがって、各クエリは、そのクエリだけで実行していた場合よりも実行に時間がかかることでしょう。リソースマネージャーにより、リソースプールを使って、リソースを使用する優先順位を決め、リソースを操作することが可能です。リソースプールは、あらかじめ割り当てられたシステムリソースのサブセットであり、関連付けられたキューがあります。本ドキュメントでは、データベースのリソースプールを作成するのに役立つ推奨ユースケースについて詳細に解説します。

カスタムリソースプールの作成

カスタムリソースプールを作成する際、設定を考慮することができるパラメーターが多数存在します。本ドキュメントに記載のないパラメーターの詳細については、Vertica ドキュメントの ALTER RESOURCE POOL を参照してください。本ドキュメントでは、次の 5 つのパラメーターに焦点を当て解説します。

- MEMORYSIZE: プールに割り当てられる初期メモリサイズ。リソースプールにこのパラメーターを設定すると、プールに対して実行されているクエリで利用可能なメモリをブロックすることができます。ただし、メモリが使用されない場合、メモリはブロックされたままであり、他のプールからのクエリには使用できません。

- MAXMEMORYSIZE: プールに対して実行されているクエリによって取得される合計メモリサイズの上限。ほとんどの場合、リソースプールは、権利が与えられたメモリのシェアを常に取得するとは限らないため、MAXMEMORYSIZE を権利が与えられたプールのメモリサイズよりも高く設定することは理にかなっています。

- MAXCONCURRENCY: プールに対して同時実行可能なクエリの最大数

- PLANNEDCONCURRENCY: プールに対して同時実行されるクエリ数の見積もり。このパラメーターは、リソースプールのクエリのバジェットを計算するために使用されます。クエリのバジェットは、クエリに割り当てられる初期メモリです。クエリの複雑さとクエリのバジェットの値に応じて、Vertica は、クエリのパフォーマンスに影響を与えるオペレーターごとの並列スレッドを割り当てます。詳細は、こちらのブログを参照してください。

- EXECUTIONPARALLELISM: リソースプールで発行された単一のクエリを処理するために使用されるスレッド数を制限

ワークロードの理解

問合せタイプ（ETL vs SELECT）、アプリケーション、ユーザーグループおよびクエリの複雑さに基づいて、ワークロードを分類することができるでしょう。詳細については、本ドキュメントの最後にある Appendix を参照してください。Appendix には、ワークロードをさらに詳しく説明する 2 つのビューの定義があります。

例：リソースプールの作成と割り当て

この例では、2 つの異なる組織について見ていきます。

- fastAnalytics：定義されたワークロードを持つ組織

- quickAnalytics：アドホックなワークロードを持つ組織

この 2 つの一般的な例を用いて、ワークロードの種類ごとにカスタムリソースプールを作成する方法を示します。

fastAnalytics という組織は、24 時間 365 日の ETL ジョブを実行しており、データをロードするために事前定義された SLA を満たしている必要があります。彼らは 10 ノード構成のクラスターを持ち、各ノードは 256GB のメモリと 48 コアを持っています。最初のステップとして、ETL および SELECT クエリに対してそれぞれ etl_pool および select_pool という 2 つのユーザーリソースプールを作成し、ETL および SELECT クエリ間でリソースを分割します。次の表は、2 つの新しいプールのリソースプールの定義を示しています。

リソースプール	MEMORYSIZE	MAXMEMORYSIZE	PLANNEDCONCURRENCY	MAXCONCURRENCY	EXECUTIONPARALLISM
general		Special95%	120		AUTO（コア数）
etl_pool		60%	48		AUTO
select_pool		60%	48		AUTO

General プールは、事前定義されたリソースプールと関連付けられていないユーザーのデフォルトプールです。事前定義されたプールがない場合、General プールに対してクエリが実行されている可能性があります。この例では、General プールのデフォルトのクエリバジェットは 5GB でした。事前定義されたプールのないユーザーは、より多くのメモリを消費していました。データベース管理者は、これらのユーザーを識別し、リソースプールを割り当てる必要があります。General プールの PLANNEDCONCURRENCY を 120 に設定し、General プールに対して実行されているクエリのメモリ使用量をより少ない 2GB のクエリバジェットを設定することもできます。

上の表は、ユーザーが作成したリソースプールのみを示しています。Metadata リソースプールを含むシステムリソースプールにおいて、各ノード上で 10GB の予約済メモリがあると仮定しました。etl_pool と select_pool のクエリバジェットはそれぞれ 3GB（MAXMEMORYSIZE/PLANNEDCONCURRENCY）です。ユーザープールを作成する利点は、SELECT クエリがメモリの 60%以上を消費できず、ETL クエリ用にメモリの約 40%を残すことができる点です。MAXMEMORYSIZE をプールに対して設定して、プールや一連のプールが他のプールを枯渇させないようにしたいとします。通常、ユーザー定義プールの MAXMEMORYSIZE の合計が使用可能な物理メモリの 20%を超えないようにすることを推奨します。

Appendix のスクリプトを使用して、ワークロードをさらに分析した後、fastAnalytics のデータベース管理者は 3 つのユーザーカテゴリを特定しました。

- ダッシュボードアプリケーションのユーザー
- データアナリスト
- 複雑なバッチデータ処理ジョブを実行しているユーザー

ダッシュボードアプリケーションユーザーは、数秒で実行されるショートクエリを実行していました。データアナリストは、数秒から 1 分未満の中規模のクエリを実行していました。バッチ処理ジョブを実行するユーザーは、1 分以上実行にかかる複雑なバッチクエリを実行していました。

この分析の後、次のステップでは、select_pool を、短いもの、中程度もの、また、複雑なクエリに合わせた 3 つのリソースプールに置き換え、それらを特定のユーザータイプに割り当てました。

- short_pool：ダッシュボードアプリケーションユーザー用

- medium_pool：データアナリスト用
- large_pool：バッチジョブを実行するユーザー用

次の表は、select_pool に代わる新しい 3 つのプールが作成された後のリソースプールの定義を示しています。

リソースプール	MEMORYSIZE	MAXMEMORYSIZE	PLANNEDCONCURRENCY	MAXCONCURRENCY	EXECUTIONPARALLISM
general		Special95%	120		AUTO（コア数）
etl_pool		40%	48		4
short_pool		40%	48		6
medium_pool		30%	18		12
large_pool		10%	3		24

この表では、デフォルト値が AUTO であるため、異なるプールの EXECUTIONPARALLELISM を任意の値で設定していることが分かります。AUTO の値は、ホスト上のコア数と同じです。fastAnalytics の場合、48 はワークロードが並列実行である場合には高すぎました。

ユーザー定義のリソースプールごとに MAXMEMORYSIZE と PLANNEDCONCURRENCY を設定して、各ユーザーのプールのメモリ消費量を制限し、サイズに基づいてプールごとに 2GB、4GB、および 8GB のクエリバジェットを設定します。etl_pool は、ライブアグリゲートプロジェクションやフラッタンテーブルがないため、クエリバジェットを 3GB、EXECUTIONPARALLELISM 値を 4 に設定しました。ライブアグリゲートプロジェクションやフラッタンテーブルを使用している場合、EXECUTIONPARALLELISM の値を 16 などと高めに設定します。

これらの初期変更が行われて数ヶ月後、fastAnalytics のデータベース管理者はこれらのクエリに対してもう一度ワークロード分析を行いました。管理者は、ワイドテーブルにデータをロードするバルク処理 ETL ジョブの中に、より長い時間メモリを消費して実行しているものがあることに気づきました。彼らのチームは、ETL ジョブを 2 つのパイプラインに分割することに決めました。1 つはトリクルロード用で、もう 1 つはワイドテーブルへのバルクロード用です。つまり、etl_pool は、各ワークロード、trickle_pool、および bulk_pool の 2 つの異なるプールに置き換えられます。これらの別々のプールでは、長時間実行される ETL ジョブは少量ロードのジョブを妨げません。

fastAnalytics では、経営陣がレポートアプリケーションを使用していました。データベース管理者は、経営陣に割り当てられるスタンドアロンのリソースプールが必要であると気づき、そのため、クエリがキューで待たされることはありませんでした。スタンドアロンのリソースプールは、General プールからメモリを借りることがないように構成されます。スタンドアロンでは、MAXMEMORYSIZE とと MEMORYSIZE が等しい値で設定されます。

次の表では、etl_pool を置き換える 2 つの新しいプールが作成され、新しい management_pool が作成された後のリソースプールの定義を示しています。

リソースプール	MEMORYSIZE	MAXMEMORYSIZE	PLANNEDCONCURRENCY	MAXCONCURRENCY	EXECUTIONPARALLISM
general		Special95%	120		AUTO（コア数）
trickle_pool		30%	28		4
bulk_pool		10%	4		4

etl_pool	40%	48	4
short_pool	40%	48	6
medium_pool	30%	18	12
large_pool	10%	3	24
management 6GB _pool	6GB	6	6

上記の表では、management_pool の MEMORYSIZE および MAXMEMORYSIZE を 6GB に設定しているため、プールに常にそのクエリに十分なメモリがあることがいえます。ただし、fastAnalytics のチームは、このプールの使用状況を定期的にチェックして、余分なメモリがこのスタンドアロンプールによってブロックされないようにする必要があります。

etl_pool に代わる trickle_pool と bulk_pool には、それぞれ 2.5GB と 6GB のクエリバジェットがあります。bulk_pool は、ワイドテーブル（250 列以上のもの）のロードに使用されます。より多くのクエリバジェットを持つワイドテーブルのロード性能を向上させることができるでしょう。

fastAnalytics は明確なワークロードを持っていましたが、一部の組織ではそうではありませんでした。たとえば、quickAnalytics の組織におけるワークロードは、完全にアドホックなものでした。quickAnalytics のデータベース管理者は、Vertica のカスケードリソースプール機能を使用して、アドホックなワークロードを対応することに決めました。

カスケードリソースプール

カスタムリソースプールを作成する際、CASCADE TO オプションを使用して、割り当てられたリソースプールの RUNTIMECAP 設定を超えるクエリを実行するセカンダリリソースプールを指定できます。

カスケードリソースプールを使用する場合、クエリは次のような内容に直面する可能性があります。

- セカンダリリソースプールには、クエリを実行するための空きメモリがありません。この場合、クエリは終了し、セカンダリプールに対して再計画され、セカンダリリソースプールのリソースに対してキューに入れられます。
- 移動対象のクエリが、セカンダリプールの MAXMEMORYSIZE より大きいメモリを必要としました。この場合、クエリの実行が終了し、セカンダリプールに対して再計画されます。
- セカンダリプールでクエリをカウントすると、MAXCONCURRENCY と同じ数になります。この場合、クエリはセカンダリリソースプールに移動せず、RUNTIMECAP を超えてもプライマリのプールで継続実行します。データコレクターテーブル DC_RESOURE_POOL_MOVE には、"Target pool does not have sufficient resources”（対象プールに十分なリソースがない）などのエラーメッセージが表示されます。

quickAnalytics のデータベースユーザーは、小、中、大のいずれかのアドホッククエリを実行しました。次に、カスケードリソースプールを実装するために使用できる 2 つの異なるモードについて説明します。

リソースプール設定モード A

設定モード A では、クエリバジェットはまったく同じですが、ランタイム上限が異なる 3 種類のリソースプールがあります。このシナリオでは、クエリがあるプールから別のプールにカスケードすると、クエリの実行は中断されませんが、メモリは新しいプールに対して処理されます。この設定は、ほとんどのクエリが短いワークロードの場合に最適です。利点は、クエリがカスケード中に再計画されないため、実行サイクルが無駄

にならないことです。ただし、まれにバジェットがより小さいプールに対する複雑なクエリは、予想よりも長く実行される可能性があります。詳細は、こちらのブログを参照してください。

同じクエリバジェットと異なるランタイム上限を持つ slow、medium、fast のプールを作成することを決定したとします。

- RUNTIMECAP が 5 秒の fast プール。この上限を超えるクエリは、medium プールにカスケードされます。
- RUNTIMECAP が 1 分の medium プール。この上限を超えるクエリは、slow プールにカスケードされます。
- RUNTIMECAP が 5 分の slow プール。この上限を超えるクエリはすべて終了します。

リソースプール	MEMORY SIZE	MAXMEMORYSIZE	PLANNEDCONCURRENCY	MAXCONCURRENCY	EXECUTION PARALLISM	RUNTIMECAP	CASCADETO
general		Special95%	AUTO（コア数）		AUTO（コア数）		
etl_pool		40%	32	32	4		
fast		50%	120	60	6	00:00:05	medium
medium		20%	10	15	12	00:01	slow
slow		10%	3	5	24	00:05	

上記の表では、fast、medium、slow のプールのクエリバジェットはそれぞれ 1GB、4GB、8GB です。PLANNEDCONCURRENCY は、これらのクエリバジェットを設定するのに役立つように設定されています。すべてのクエリは fast リソースプールに対して計画され、カスケード後に再計画されないようにするため、クエリバジェットを 1GB で実行します。medium および slow のプールのバジェットが高くなるため、これらのプールに割り当てられたユーザーが利用できるようになります。

MAXCONCURRENCY はリソースプールで設定され、プールに対する同時クエリの数を制限します。

リソースプール設定モード B

設定モード B では、かなり異なるクエリバジェット、最大並列実行性、および実行時上限を持つ 3 種類のリソースプールを設定します。この設定では、クエリがあるプールから別のプールにカスケードされると、クエリの実行が終了し、新しいプールに対してより高いバジェットでクエリが再計画されます。この設定は、同じユーザーが混在したワークロードをシンプル、中程度、複雑なクエリで実行している場合に最適です。リソースプールの実行時上限を小さな値に設定することにより、リソースプールの移動によって失われる実行サイクルを最小限に抑えます。クエリは、より適切なバジェットでプール上で再計画され、再計画で失われた時間を補うためにより速く実行されます。

quickAnalytics のデータベースユーザーは、小、中、大のいずれかのアドホッククエリを実行しました。管理者は、それぞれ 1GB、2GB、5GB のクエリバジェットで、fast、medium、slow のプールを作成しました。彼らはプールの実行時制限を次のように設定しました:

- RUNTIMECAP が 3 秒の fast プール。この上限を超えるクエリは、medium プールにカスケードします。

- RUNTIMECAP が 15 秒の medium のプール。この上限を超えるクエリは、slow プールにカスケードします。

- RUNTIMECAP が 1 分の slow プール。この上限を超えるクエリはすべて終了します。

SELECT クエリを実行しているすべてのユーザーが fast プールに割り当てられました。設定パラメーター CascadeResourcePoolAlwaysReplan を 1 に設定する必要があります。値が 1 の場合、クエリが新しいリソースプールに移動されると、クエリは再計画されます。また、ユーザーは 3 つのリソースプールすべてに対して使用権限を持っている必要があります。

注意：CascadeResourcePoolAlwaysReplan パラメーターは、Vertica の製品ドキュメントには記載されていません。

リソースプール	MEMORY SIZE	MAXMEMORYSIZE	PLANNEDCONCURRENCY	MAXCONCURRENCY	EXECUTION PARALLISM	RUNTIMECAP	CASCADETO
general		Special95%	AUTO（コア数）		AUTO（コア数）		
etl_pool		40%	32	32	4		
fast		50%	120	60	6	00:00:03	medium
medium		20%	20	10	12	00:00:15	slow
slow		10%	5	5	24	00:05	

リソースマネージャー分析

リソースプールの設計者は、次のことを行う必要があります。

- クエリがリソースキューでリソースの待機状態にない、あるいは、最小限の待機時間であることを確認します。以下のクエリを使用すると、リソースキューの現在の状態と、個々の履歴クエリがキュー内で費やした時間を確認できます。

```
=> SELECT* FROM resource_acquisitions;
```

- リソースプールのバジェットを細かく調整して、クエリのパフォーマンスを向上させます。次のクエリは、他の重要なリソースプールパラメーターとともに、各プールのクエリのバジェットを示しています。

```
=> SELECT pool_name, memory_size_kb, max_memory_size_kb,
planned_concurrency, max_concurrency,
query_budget_kb FROM resource_pool_status WHERE node_name ilike
'%0001%' ORDER BY query_budget_kb desc;
```

- 大量のメモリを消費したり、異常な実行時間で実行されている不定期の不正なクエリを特定します。これが他の同時実行クエリに影響していることが考えられるためです。次の例は、25GB を超えるトランザクションを示しています。

```
=> SELECT * FROM user_workload WHERE mem_kb > '25*1024^2' ORDER BY
query_duration_us desc;
```

マネージメントコンソールを使用してリソースプールをモニタリングすることも可能です。詳細については、Vertica documentation を参照してください。

Appendix

ワークロードを分析するには、次の 2 つのビューを使用して、データベース管理者としてクエリを作成および実行します。

USER_WORKLOAD ビューは、データコレクターテーブルの dc_resource_acquisitions、dc_requests_issued および dc_requests_completed テーブルを結合して、クエリタイプ、使用されているリソースプール、クエリで使用されているメモリー、およびデータベースユーザーによって実行されたクエリのクエリ実行時間をマイクロ秒単位で取得します。このビューには、大量のメモリを消費するクエリを識別するために使用できるトランザクション ID とステートメント ID、およびそのクエリを発行したユーザーも含まれています。

USER_WORKLOAD_BY_HOUR は、1 時間ごとに集計される USER_WORKLOAD ビューの最上部のビューです。これには、ユーザーごとに実行されるショート、メディアム、およびロングクエリが含まれ、平均実行時間、最小、最大および平均メモリーが 1 人のユーザーあたり 1 キロバイト単位で集計されます。実行時間が 10 秒未満、実行時間が 5 分以上、実行時間が 10 秒から 5 分であると仮定しています。

```
=> CREATE OR REPLACE VIEW USER_WORKLOADAS
SELECT ri.user_name,
ri.time,
ri.request_type,
ra.pool_name,
ra.transaction_id,
ra.statement_id,
(memory_kb) mem_kb,

extract(epoch from rc.time - ri.time) * 1e6 as query_duration_us
FROM
(SELECT transaction_id,
statement_id,
pool_name,
max(memory_kb) memory_kb
FROM dc_resource_acquisitions
WHERE request_type!='Acquire' and pool_name <> 'sysquery'
GROUP BY 1,2,3)ra
JOIN dc_requests_issued AS ri ON ra.transaction_id=ri.transaction_id
AND ra.statement_id=ri.statement_id
JOIN dc_requests_completed as rc on rc.session_id=ri.session_id
AND rc.request_id=ri.request_id
AND rc.node_name=ri.node_name;
```

```
=> CREATE OR REPLACE VIEW USER_WORKLOAD_BY_HOUR
AS
SELECT
TIME::varchar(13),
user_name,
```

```
pool_name,
CASE
WHEN avg(query_duration_us) > 300000000 THEN 'LONG_QUERY'
WHEN avg(query_duration_us) < 10000000 THEN 'SHORT_QUERY'
ELSE 'MEDIUM_QUERY'
END AS query_size,
count(*) query_count,
avg(query_duration_us)::int avg_qd_us,
min(mem_kb)::int min_mem_kb,
avg(mem_kb)::int avg_mem_kb,
max(mem_kb)::int max_mem_kb
FROM USER_WORKLOAD
GROUP BY 1,2,3
ORDER BY 1,2,3,4;
```