Alibaba Cloud Table Store

ベストプラクティス

Document Version20190523

目次

1 テーブル操作

ここでは、Alibaba Cloud Table Store の使用を最適化するための推奨事項について詳しく説明します。

適切に設計されたプライマリキー

Table Store はパーティションキーに応じてテーブルデータを動的にパーティションに分割し、各パーティションは 1 つのサーバーノードでホストされます。 パーティションキー値は、最小のパーティション単位です。 同じパーティションキー値のデータは分割できません。 この場合、アプリケーションは Table Store の機能を活用するために、パーティション間でデータ分散とアクセス分散のバランスをとる必要があります。

Table Store は、テーブル内の行をプライマリキーでソートします。 適切に設計されたプライマリキーを使用すると、Table Store の高いスケーラビリティを最大限に活用しながら、パーティション間でのデータ分散のバランスを良くすることができます。

パーティションキーを選択するときは、次の点に注意してください。

- 1 つのパーティションキー値のすべての行のデータは 10 GB を超えることはできません。

  
  注 :
  10 GB は厳しい制限ではありませんが、ホットスポットを避けるために推奨します。
- 同じテーブルの異なるパーティションキー値のデータは論理的に独立しています。
- アクセス負荷を狭い範囲の連続したパーティションキー値に集中させないでください。

例

学生証を使って生徒の購買の記録を保存するテーブルがあるとします。 このシナリオでは、

- 各学生証は 1 つの CardID に対応します。
- 各販売者は 1 つの SellerID に対応しています。
- 各POS端末は、グローバルに固有の DeviceID に対応します。
- POS 端末によって生成された各購入に対して、1 つの OrderNumber が記録されます。 デバイスによって生成された OrderNumber はデバイスに固有ですが、グローバルに固有ではありません。

たとえば、異なるPOS端末が、同じ OrderNumber を使用して 2 つの別々の購入記録を生成することがあります。 同じ POS 端末によって生成された各 OrderNumber は、異なるタイムスタ

ンプを持ちます。 新しい購入記録は、古い購入記録よりも大きい連続した OrderNumbers を持ちます。 すべての購入記録はリアルタイムでテーブルに書き込まれます。

Table Store の使用を最適化するには、CardID または DeviceID をテーブルのパーティションキーとして使用することを推奨します。

- CardID を使用することを強く推奨します。一般的に、各カードの購入記録の数は毎日同じであるため、各パーティションのアクセス負荷はバランスが取れています。 これにより、予約済み読み書きスループットを効率的に利用できます。
- 1 日あたりの各販売者の購入記録の数は異なりますが、1 日あたりの各購入デバイスによって生成される購入記録の数は概算できるため、DeviceID を使用することを推奨します。 この見積もりは、レジの注文処理速度に基づいて計算されます。これは、1 日に購入デバイスによって生成される購入記録の数を決定します。 したがって、DeviceID は、アクセス負荷のバランスの取れた分散を保証するテーブルのパーティションキーとして適しています。

SellerID と OrderNumber を使用することは推奨しません。 SellerID は、利用可能な販売者数が限られていることを示しているため推奨しません。したがって、少数の販売者が多数の購入記録を生成するシナリオでは、各パーティションに対するアクセス負荷のバランスを取るのに役立ちません。 OrderNumber は、同時に生成された購買発注の連続的な増加のために推奨しません。その結果、同じ期間にグループ化された注文が発生します。 これにより、読み書きスループットの有効性が制限されます。

 注 :
OrderNumber をパーティションキーにする必要がある場合は、それをハッシュし、結果のハッシュ値を OrderNumber プレフィックスとして使用できます。 このプロセスにより、データおよびアクセス負荷の均等な分散が可能になります。

結合されたパーティションキー

Table Store の使用を最適化するために、単一パーティションキー値のデータ量が 10 GB を超えないようにすることを推奨します。 単一のテーブルパーティションキー値に含まれるすべての行の合計データ量が 10 GB を超える場合は、テーブルを設計するときに複数の元のプライマリキー列を 1 つのパーティションキーに結合できます。

例

前述の学生証の購入記録の例のように、プライマリキー列が [DeviceID, SellerID, CardID, OrderNumber] であるとします。 DeviceID は、このテーブルのパーティションキーであり、単一の DeviceID のすべての行からの合計データ量は 10 GB を超えることがあります。 この場

合、テーブルの最初のプライマリキー列 (パーティションキー) として DeviceID、SellerID および CardID を結合します。

元の表は次のとおりです。

DeviceID	SellerID	CardID	OrderNumber	attrs
16	'a100'	66661	200001	...
54	'a100'	6777	200003	...
54	'a1001'	6777	200004	...
167	'a101'	283408	200002	...

パーティションキーを作成するために、 DeviceID、SellerID および CardID を結合した後、新しいテーブルは次のように表示されます。

CombineDeviceIDSellerIDCardID	OrderNumber	attrs
'16:a100:66661'	200001	...
'167:a101:283408'	200002	...
'54:a1001:6777'	200004	...
'54:a100:6777'	200003	...

元のテーブルでは、DeviceID = 54 の 2 行は、同じパーティションキー値 54 の 2 つの購入記録に属しています。 新しく作成されたテーブルでは、これら 2 つの購入記録のパーティションキーは異なります。複数のブラいまりキー列を結合してパーティションキーを形成すると、テーブル内の各パーティションキーの合計データ量を減らすことができます。

プライマリキー列を結合してテーブルを形成すると、いくつかの欠点があります。 DeviceID は整数型のプライマリキー列です。 元の表では、DeviceID = 54 の購入記録が DeviceID = 167 の購入記録の前にリストされています。 最初の 3 つのプライマリキー列を文字列型のプライマリキー列に結合した後、DeviceID = 54 の購入記録が DeviceID = 167 の購入記録の後に表示されます。 アプリケーションが DeviceID の範囲 [15, 100) からすべての購入記録を読み取る必要がある場合、上記のテーブルは最適ではありません。

この状況に対処するために、DeviceID の前に 0 を追加できます。 追加する 0 の数は、DeviceID の最大桁数によって決まります。 DeviceID の範囲が [0, 999999] の場合は、すべての DeviceID が 6 桁になるように 0 を追加してから結合します。 結果はとして得られるテーブルは以下のとおりです。

CombineDeviceIDSellerIDCardID	OrderNumber	attrs
'000016:a100:66661'	200001	...
'000054:a1001:6777'	200004	...
'000054:a100:6777'	200003	...
'000167:a101:283408'	200002	...

ただし、ID の前にゼロを埋め込んだ後でも、テーブルはまだ完全には最適化されていません。これは、DeviceID = 54 の 2 行のためです。 SellerID = 'a1001' の行は、SellerID = 'a100' の後に表示されます。 この矛盾は、コネクタとしての `:` が原因で発生します。これは、'a1001' が 'a101' よりも大きいにもかかわらず、‘000054:a1001’ が ‘000054:a100:’ が辞書的に小さいという辞書式順序に影響を与えます。 この問題を解決するには、他の使用可能なすべての文字の ASCII コードより小さい文字を選択してください。 このテーブルでは、SellerID値 は大文字と小文字、および数字を使用します。 `,` をコネクタとして使用することを推奨します。`,` の ASCII コードは、SellerID に使用できるすべての文字よりも小さいからです。

`,` を使用してから結合すると、次のような最適化されたテーブルが作成されます。

CombineDeviceiDSellerIDCardID	OrderNumber	attrs
'000016,a100,66661'	200001	...
'000054,a100,6777'	200003	...
'000054,a1001,6777'	200004	...
'000167,a101,283408'	200002	...

パーティションキーを接続して作成された上記のテーブルでは、記録の順序は元のテーブルのものと一致しています。

まとめ

単一のパーティションキー値のすべての行の合計データサイズが 10 GB を超える場合は、複数のプライマリキー列を結合してパーティションキーを形成し、個々のパーティションキー値のデータサイズを最小化できます。 パーティションキーを結合するときは、次の点に注意してください。

- 結合するプライマリキー列を選択するときは、結合後に同じパーティションキーの元の行のパーティションキーが異なることを確認してください。

- 整数型のプライマリキー列を接合するときは、数字の前に 0 を追加して行の順序が変わらないようにします。
- 結合部を選択するときは、新しいパーティションキーの辞書式順序への影響を考慮してください。 理想的な方法は、他のすべての使用可能な文字より小さい ASCII コードを持つ結合部を選択することです。

パーティションキーへのハッシュプレフィックスの追加

例

適切に設計されたプライマリキーセクションで、OrderNumber をテーブルのパーティションキーとして使用しないことを推奨します。 OrderNumber は順次増加するので、購入記録は常に最新の OrderNumber 範囲で書き込まれます。つまり、以前の OrderNumber 範囲では書き込み負荷がかかりません。 これはアクセス負荷の不均衡を引き起こし、予約済み読み書きスループットの非効率的な使用をもたらします。 順次増加するキー値をパーティションキーとして使用する必要がある場合は、ハッシュプレフィックスをパーティションキーに結合します。 このようにして、OrderNumber はテーブル全体にランダムに分散され、アクセス負荷の分布が安定します。

パーティションキーとして OrderNumber を使用する購買記録テーブルは次のとおりです。

OrderNumber	DeviceID	SellerID	CardID	attrs
200001	16	'a100'	66661	...
200002	167	'a101'	283408	...
200003	54	'a100'	6777	...
200004	54	'a1001'	6777	...
200005	66	'b304'	178994	...

例として、OrderNumbers の場合は、md5 アルゴリズムを使用してプレフィックスを計算し(他のハッシュアルゴリズムも使用できます)、それを結合して HashOrderNumber を作成できます。 md5 アルゴリズムで計算されたハッシュ文字列は長すぎる可能性があるので、最初の数桁だけを使用して、連続した OrderNumbers の記録をランダムに分布させることができます。この例では、最初の 4 桁を使用して次のテーブルを作成しています。

HashOrderNumber	DeviceID	SellerID	CardID	attrs
'2e38200004'	54	'a1001'	6777	...
'a5a9200003'	54	'a100'	6777	...

HashOrderNumber	DeviceID	SellerID	CardID	attrs
'c335200005'	66	'b304'	178994	...
'db6e200002	167	'a101'	283408	...
'ddba200001'	16	'a100'	66661	...

後で購入記録にアクセスするときは、同じアルゴリズムを使用して OrderNumber のハッシュプレフィックスを計算し、購入記録に対応する HashOrderNumber を取得します。 パーティションキーにハッシュプレフィックスを追加することの 1 つの欠点は、元々連続していたレコードが分散されることです。 つまり、GetRange 操作を使用して、論理的に連続した一連のレコードを取得することはできません。

データを並行して書き込む

Table Store テーブルを複数のパーティションに分割すると、これらのパーティションは複数のテーブルストアサーバーに分散されます。 データのバッチが Table Store にアップロードされるようにプライマリキーによって順序付けられている場合、データが同じ順序で書き込まれると、書き込み負荷が特定のパーティションに集中する可能性があります。 他のパーティションはアイドル状態のままですが、このパーティションには高い負荷がある可能性があります。 この操作では、予約済み読み書きスループットが十分に活用されず、データのインポート速度が低下する可能性があります。

この問題を解決するには、次のいずれかの方法を使用してデータのインポート速度を上げます。

- 元のデータ順序を中断してからインポートします。 書き込まれたデータが各パーティションに均等に分散されていることを確認してください。
- 並列データインポートに複数のワーカースレッドを使用します。 大きなデータセットを複数の小さなセットに分割します。 次に、ワーカースレッドは、インポートする小さいセットをランダムに選択します。

コールドデータとホットデータを区別する

時間管理データの管理を誤ると問題が発生する可能性があります。 前述の学生の取引記録の例を使用すると、アプリケーションは頻繁に最新の記録を照会し、最新の記録に基づいて統計を処理して編集するため、一部の購入記録はより高いアクセス確率を持つ可能性があります。 しかし、古い購入記録は保管スペースを占め続け、コールドになります。 大量のコールドデータがテーブルに含まれている場合 (登録されなくなってもシステムに保持されている学生の CardID など)、予約済み読み書きスループットが有効にに利用されず、パーティション全体のアクセス負荷が不均衡になります。

時間に敏感なデータを効果的に管理するには、異なるテーブルを使用してコールドデータとホットデータを分けます。 それぞれに異なる予約済み読み書きスループットを設定します。 たとえば、購入記録は月ごとに異なるテーブルに分割し、新しいテーブルが月ごとに作成します。 予約済み読み書きスループットは、各テーブルに対して次のように設定できます。

- アクセスニーズを満たすために (新しい購入記録はレガシーデータよりも照会される可能性が高い)、今月の最新の購入記録を持つテーブルに対して高い予約済み読み書きスループットを設定できます。
- 低い予約済み書き込みスループットと高い予約済み読み取りスループットは、新しいデータがほとんど書き込まれないテーブル(過去数ヶ月間) に対して設定できますが、クエリは引き続き実行されます。
- メンテナンス期間 (1 年以上の履歴レコードなど) を超えたテーブルには、予約済み読み書きスループットを低く設定できます。 その後、これらのテーブルをエクスポートして OSS アーカイブに復元するか、削除することができます。

2 データ操作

ここでは、Table Store のデータ操作を最適化するための推奨事項について説明します。 特に、属性列とアプリケーション要求エラーを効果的に管理する方法について詳しく説明します。

属性列間で表を分割

テーブルの行に多数の属性列があるが、各操作がこれらの列の一部にしかアクセスしない場合は、テーブルを複数に分割できます。 異なるアクセス頻度の属性列は、異なるテーブルに配置できます。

たとえば、商品の数量、商品の価格、商品の説明を含む行を含む商品管理システムでは、次のようになります。

- アイテムの数量と価格は、ストレージ領域をほとんど消費しない整数型の値ですが、頻繁に変更されます。
- アイテムの説明は String 型の値で、より多くのストレージ領域を消費しますが、変更はほとんどされません。

大部分の操作ではアイテムの数量と価格の整数型の値を更新するだけでよいため、テーブルを 2 つのテーブルに分割し、一方にこれらの 2 つの値を、もう一方に String 型のアイテムの説明を含めることができます。

テキストベースの属性列を圧縮

属性列に大量のテキストが含まれている場合は、属性列を圧縮して Table Store にバイナリ型として格納できます。 このプロセスにより、スペースを節約し、アクセス操作で消費される容量ユニットを削減して、Table Store の使用コストを削減します。

OSS に属性列を格納

Table Store では、単一の属性列のサイズが 2 MB に制限されています。 2 MB を超えるファイルを保存する必要がある場合は、Alibaba Cloud Object Storage Service (OSS) を使用することを推奨します。 OSS は、Alibaba Cloud Table Store と比較して低コストで大きなファイルを保存できる代替のストレージサービスです。

OSS を使用できない場合は、値が 2 MB を超える属性列を複数の小さい行に分割してから Table Store に格納できます。

エラー再試行間隔を追加

アプリケーションの要求が失敗して再試行エラーが返された場合は、しばらくしてから要求を再試行することを推奨します。 ベストプラクティスとして、無作為化または指数関数的に増加するバックオフは、雪崩効果を回避するのに役立ちます。