Red Hat Data Grid 8.4

Data Grid のパフォーマンスとサイジングガイド

Data Grid のデプロイメントのプランニングおよびサイジング

Last Updated: 2024-04-30

Red Hat Data Grid 8.4 Data Grid のパフォーマンスとサイジングガイド

Data Grid のデプロイメントのプランニングおよびサイジング

法律上の通知

概要

デプロイメントの計画は、Red Hat Data Grid のユースケースを調査し、ニーズに合ったアーキテクチャーを決定することから始まります。そこから、データの耐久性と遅延の増加など、Data Grid 機能のパフォーマンスに関するさまざまな考慮事項とトレードオフを検討する必要があります。初期デプロイメントを開始して実行すると、Data Grid クラスターのベンチマークを開始し、パフォーマンスの期待値を検証する準備が整います。

目次

RED HAT DATA GRID

Data Grid は、高性能の分散型インメモリーデータストアです。

スキーマレスデータ構造

さまざまなオブジェクトをキーと値のペアとして格納する柔軟性があります。

グリッドベースのデータストレージ

クラスター間でデータを分散および複製するように設計されています。

エラスティックスケーリング

サービスを中断することなく、ノードの数を動的に調整して要件を満たします。

データの相互運用性

さまざまなエンドポイントからグリッド内のデータを保存、取得、およびクエリーします。

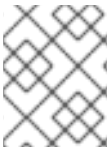
DATA GRID のドキュメント

Data Grid のドキュメントは、Red Hat カスタマーポータルで入手できます。

- Data Grid 8.4 ドキュメント
- Data Grid 8.4 コンポーネントの詳細
- Data Grid 8.4 でサポートされる設定
- Data Grid 8 機能のサポート
- Data Grid で非推奨の機能

DATA GRID のダウンロード

Red Hat カスタマーポータルで Data Grid Software Downloads にアクセスします。



注記

Data Grid ソフトウェアにアクセスしてダウンロードするには、Red Hat アカウントが必要です。

多様性を受け入れるオープンソースの強化

Red Hat では、コード、ドキュメント、Web プロパティーにおける配慮に欠ける用語の置き換えに取り組んでいます。まずは、マスター (master)、スレーブ (slave)、ブラックリスト (blacklist)、ホワイトリスト (whitelist) の 4 つの用語の置き換えから始めます。この取り組みは膨大な作業を要するため、今後の複数のリリースで段階的に用語の置き換えを実施して参ります。詳細は、Red Hat CTO である Chris Wright のメッセージ をご覧ください。

第1章 DATA GRID のデプロイメントモデルとユースケース

Data Grid は、さまざまなユースケースをサポートする柔軟なデプロイメントモデルを提供します。

- Red Hat Build of Quarkus、Red Hat JBoss EAP、および Spring アプリケーションのパフォーマンスを大幅に改善します。
- サービスの可用性および継続性を確保します。
- 運用コストの削減。

1.1. DATA GRID デプロイメントモデル

Data Grid には、キャッシュ用にリモートと組み込みの 2 つのデプロイメントモデルがあります。どちらの導入モデルでも、アプリケーションは、従来のデータベースシステムと比較して、読み取り操作の待ち時間を大幅に短縮し、書き込み操作のスループットを向上させてデータにアクセスできます。

リモートキャッシュ

Data Grid Server ノードは、専用の Java 仮想マシン (JVM) で実行されます。クライアントは、Hot Rod、バイナリー TCP プロトコル、または REST over HTTP を使用してリモートキャッシュにアクセスします。

埋め込みキャッシュ

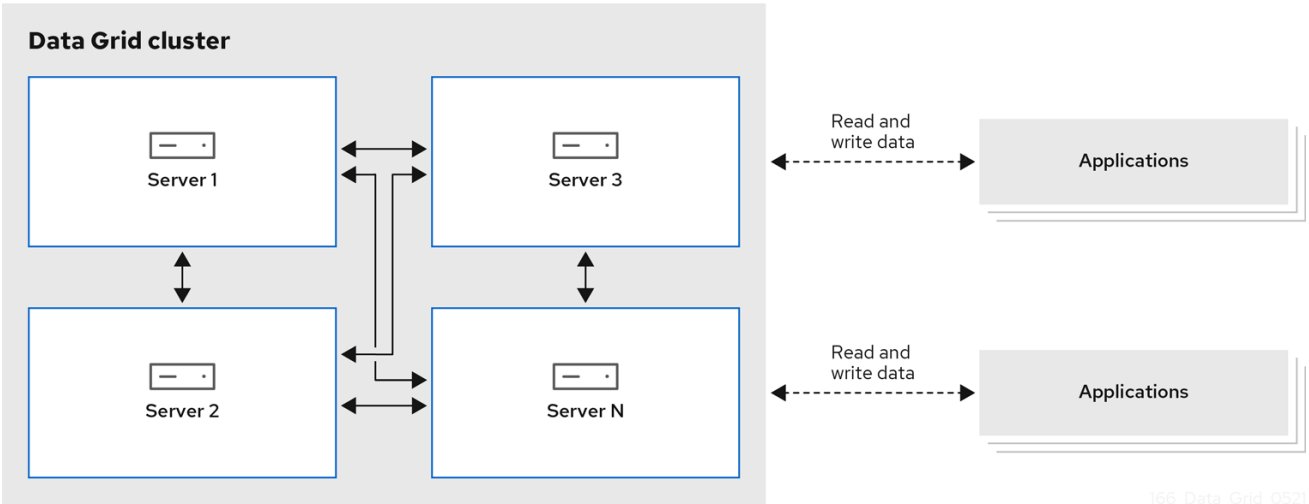
Data Grid は、Java アプリケーションと同じ JVM で実行されます。つまり、データはコードが実行されるメモリー領域に保存されます。

Red Hat は、ほとんどのデプロイメントにはサーバー/クライアントアーキテクチャーを推奨します。データレイヤーはビジネスロジックから分離されているため、リモートキャッシュでは、デプロイメントがより高速になります。Data Grid Server は、開発コストを削減できるように、監視および可観測性やその他の組み込み機能を提供します。

ニアキャッシュ

ニアキャッシング機能により、リモートクライアントはデータをローカルに保存できます。つまり、読み取りが集中するアプリケーションは、呼び出しごとにネットワークを横断する必要がありません。ニアキャッシュにより読み取り操作の速度が大幅に増大し、組み込みキャッシュと同じパフォーマンスを実現します。

図1.1 リモートキャッシュデプロイメントモデル



1.1.1. プラットフォームおよび自動化ツール

必要な QoS を実現すると、CPU および RAM リソースが最適な Data Grid を提供することになります。リソースの数が多すぎると、ホストリソースが過剰に使用されている間、Data Grid のパフォーマンスのダウングレードによりコストが迅速に増大します。

CPU または RAM の適切な割り当てを見つけるために Data Grid クラスターをベンチマークおよびチューニングしますが、クラスターをスケーリングし、リソースを効率的に管理するために適切な種類の自動化ツールを提供するホストプラットフォームも考慮する必要があります。

ベアメタルまたは仮想マシン

Red Hat Ansible と Data Grid の設定を管理し、サービスをポーリングして、可用性を確保し、リソースの使用が最適となるように、RHEL または Microsoft Windows を組み合わせます。

Automation Hub から入手できる Data Grid の Ansible コレクション は、クラスターのインストールを自動化し、Keycloak 統合とクロスサイトレプリケーションのオプションを含みます。

OpenShift

Kubernetes オーケストレーションを活用して、Pod を自動的にプロビジョニングし、リソースに制限を課し、ワークロードの需要に合わせて Data Grid クラスターを自動的にスケーリングします。

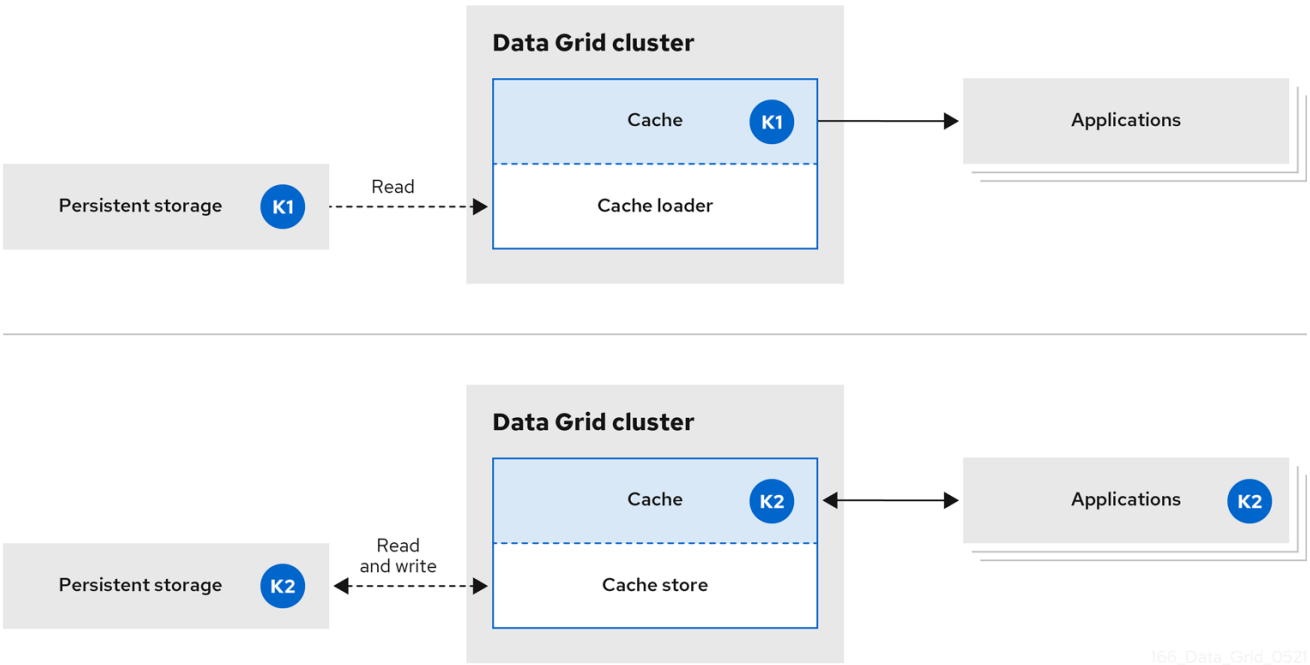
1.2. インラインキャッシュ

Data Grid は、永続ストレージにあるデータのすべてのアプリケーション要求を処理します。

インラインキャッシュでは、Data Grid はキャッシュローダーとキャッシュストアを使用して、永続ストレージのデータを操作します。

- キャッシュローダーは、永続ストレージへの読み取り専用アクセスを提供します。
- キャッシュストアは永続ストレージへの読み取りおよび書き込みアクセスを提供します。

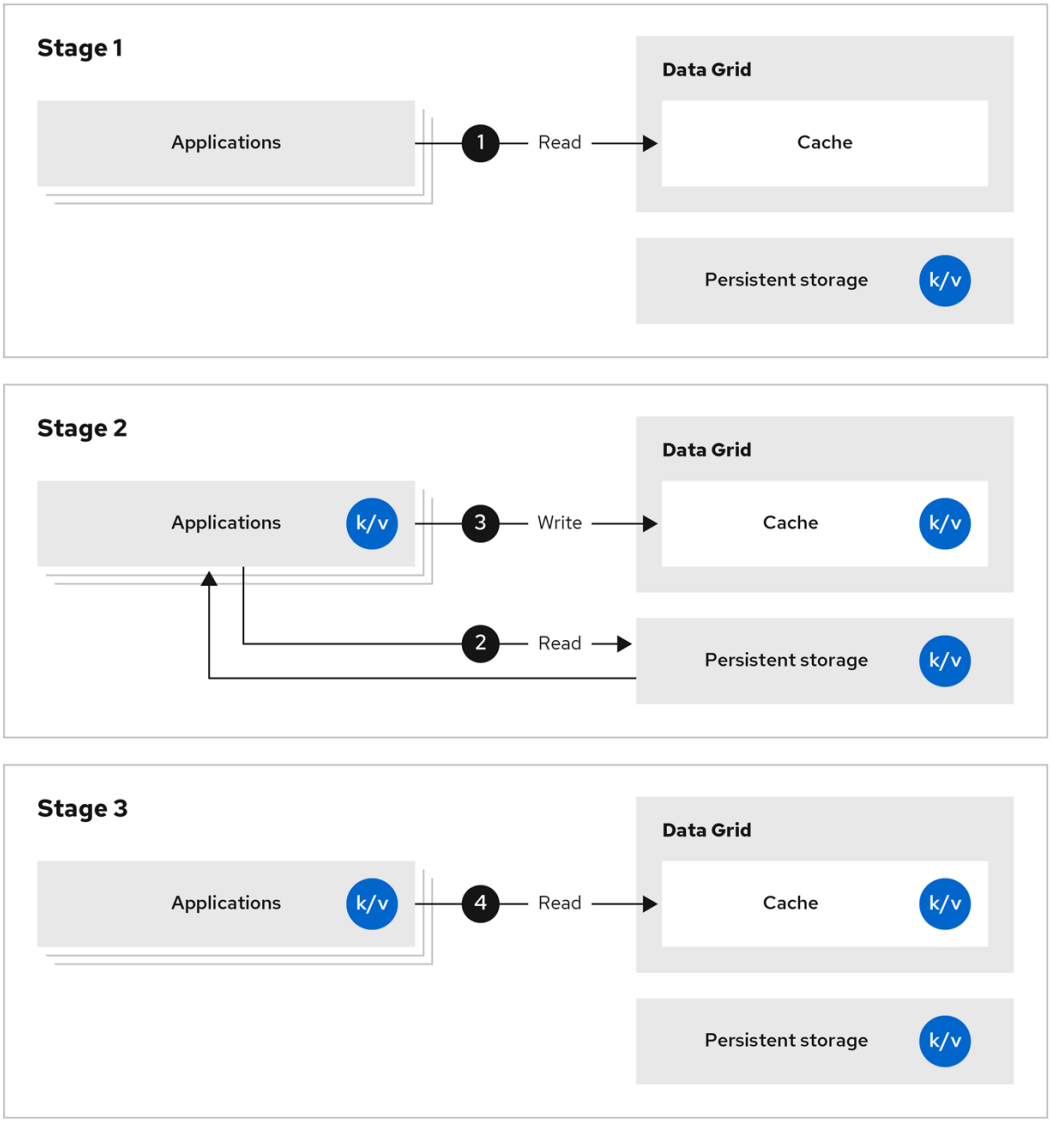
図1.2 インラインキャッシュ



1.3. サイドキャッシュ

Data Grid は、アプリケーションが永続ストレージから取得するデータを保存し、永続ストレージに対して読み取り操作の数を減らし、後続の読み取りの応答時間を増やします。

図1.3 サイドキャッシュ



サイドキャッシュでは、アプリケーションは永続ストレージから Data Grid クラスターにデータを追加する方法を制御します。アプリケーションがエントリーを要求すると、以下が発生します。

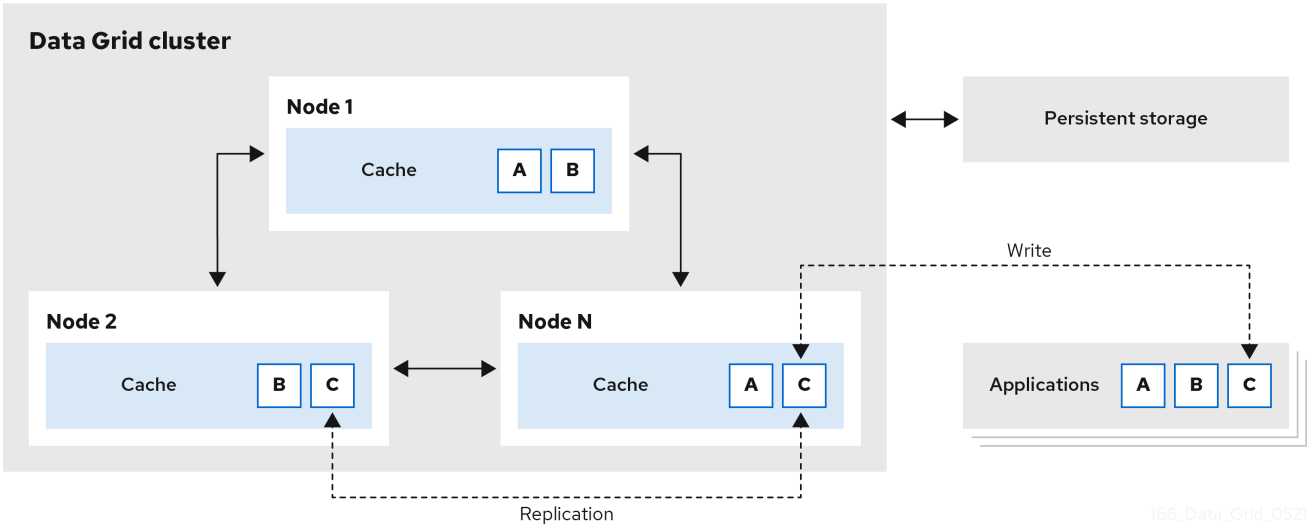
1. 読み取り要求は Data Grid に送信されます。
2. エントリーがキャッシュにない場合、アプリケーションは永続ストレージから要求します。
3. アプリケーションはエントリーをキャッシュに配置します。
4. アプリケーションは次の読み取りの Data Grid からエントリーを取得します。

1.4. 分散メモリー

Data Grid は、一貫性のあるハッシュ技術を使用して、クラスター全体でキャッシュ内の各エントリーの固定のコピーを保存します。分散キャッシュは、データ層を線形にスケールでき、容量をノードに参加として増やすことができます。

分散キャッシュは、Data Grid クラスターに冗長性を追加し、耐障害性と耐久性の保証を提供します。Data Grid デプロイメントは通常、正常なシャットダウンおよびバックアップから復元するためにクラスターの状態を保持するように永続ストレージとの統合を設定します。

図1.4 分散キャッシュ



1.5. セッションの外部化

Data Grid は、Red Hat Build of Quarkus、Red Hat JBoss EAP、Red Hat JBoss Web Server、および Spring 上に構築されたアプリケーションに外部キャッシュを提供できます。これらの外部キャッシュは、HTTP セッションやその他のデータはアプリケーションレイヤーとは別に保存されます。

Data Grid にセッションを外部化すると、以下の利点があります。

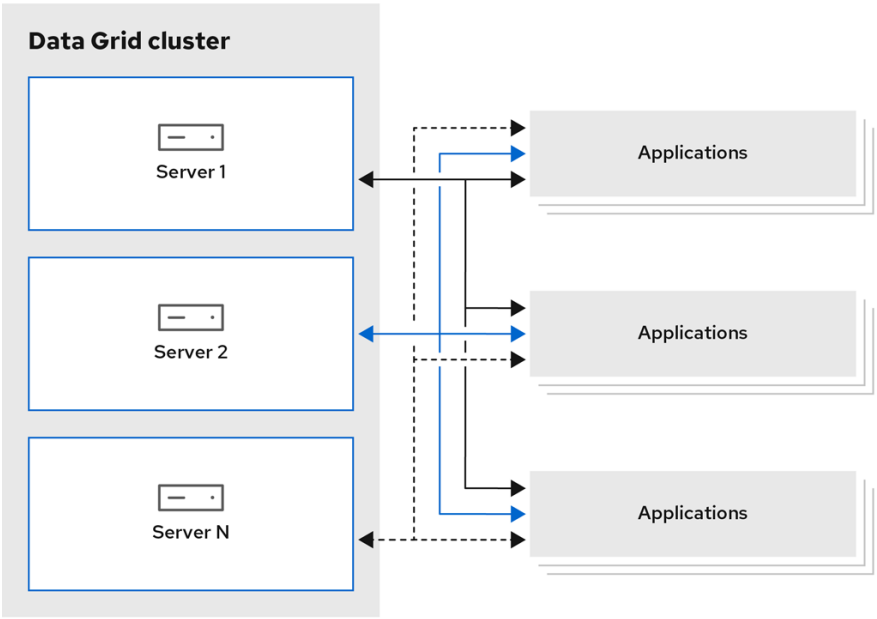
エラスティシティ

アプリケーションをスケーリングする際に操作を再調整する必要がなくなります。

メモリーフットプリントが小さくなります。

セッションデータを外部キャッシュに格納すると、アプリケーションの全体的なメモリー要件が軽減されます。

図1.5 セッションの外部化

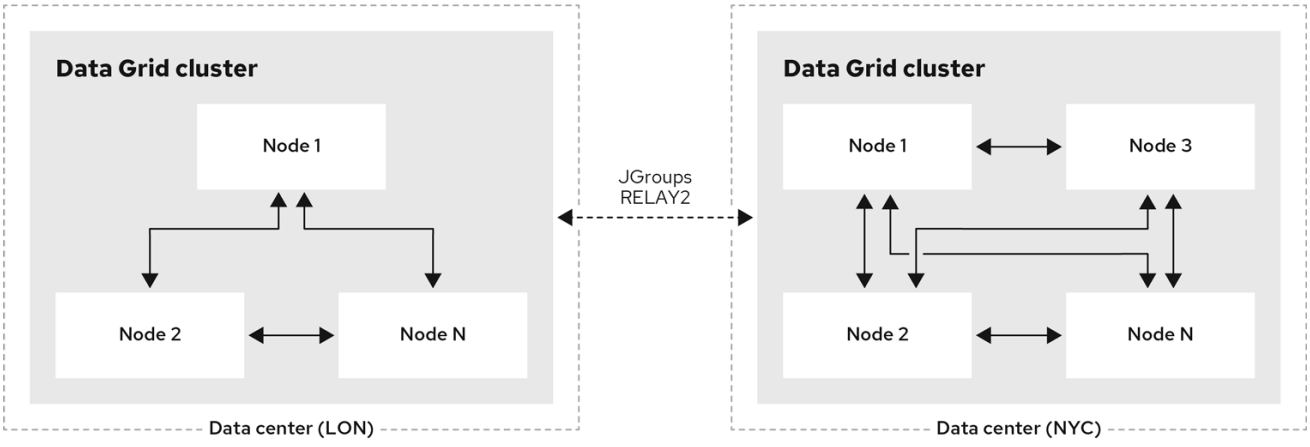


1.6. クロスサイトレプリケーション

Data Grid は、地理的に分散したデータセンターで実行されているクラスター間および異なるクラウドプロバイダー間でデータをバックアップできます。クロスサイトレプリケーションは、Data Grid にグローバルクラスタービューを提供し、以下を提供します。

- 停電や災害時のサービス継続を保証します。
- グローバルに分散されたキャッシュ内のデータへの単一アクセスポイントをクライアントアプリケーションに提供します。

図1.6 クロスサイトレプリケーション



第2章 DATA GRID デプロイメントのプランニング

Data Grid デプロイメントのパフォーマンスを最適化するには、以下を実行する必要があります。

- データセットのサイズを計算する。
- ユースケースと要件に最も適しているクラスター化キャッシュモードのタイプはどれかを判断する。
- 耐障害性と一貫性を保証する Data Grid 機能のパフォーマンストレードオフと考慮事項を理解する。

2.1. パフォーマンスメトリックに関する考慮事項

Data Grid には非常に多くの設定可能な組み合わせが含まれているため、すべてのユースケースに対応するパフォーマンスメトリックの式を 1 つに決定することは不可能です。

Data Grid パフォーマンスおよびサイジングガイド ドキュメントの目的は、Data Grid デプロイメントの要件を決定するのに役立つユースケースとアーキテクチャーに関する詳細を提供することです。

さらに、次の相互に関連する要因を考慮し、Data Grid に適用してください。

- クラウド環境で使用可能な CPU およびメモリーリソース
- 並行して使用されるキャッシュ
- Get、put、query の分散
- ピーク負荷とスループットの制限
- データセットでのクエリーの制限
- キャッシュあたりのエントリー数
- キャッシュエントリーのサイズ

さまざまな組み合わせと未知の外部要因の数を考えると、すべての Data Grid のユースケースを満たすパフォーマンス計算を提供することは不可能です。前述の要因のいずれかが異なる場合、あるパフォーマンステストを別のテストと比較することはできません。

パフォーマンスメトリックを限定して収集する Data Grid CLI を使用して、基本的なパフォーマンステストを実行できます。要件にあった結果をテストで出力されるように、パフォーマンステストをカスタマイズできます。テスト結果は、Data Grid キャッシング要件の設定とリソースを決定するのに役立つベースラインメトリックを提供します。

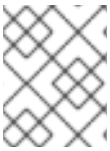
現在の設定のパフォーマンスを測定し、要件を満たしているかどうかを確認します。要件が満たされていない場合は、設定を最適化し、パフォーマンスを再測定してください。

2.2. データセットのサイズを計算する方法

Data Grid デプロイメントのプランニングでは、データセットのサイズを計算してから、適切なノード数と、データセットを保持する RAM の容量を測定する必要があります。

以下の式を使用して、データセットの合計の概算サイズを予測できます。

```
Data set size = Number of entries * (Average key size + Average value size + Memory overhead)
```



注記

リモートキャッシュでは、マーシャリング形式で鍵のサイズと値サイズを計算する必要があります。

分散キャッシュのデータセットサイズ

分散キャッシュには、データセットサイズを判断するために追加の計算が必要です。

通常の運用条件では、分散キャッシュは、設定する **所有者の数** と等しい各キー/値エントリーのコピーを多数保存します。クラスターのリバランス操作時に、追加のコピーがあるエントリーもあるので、**所有者数 + 1** を算出してそのシナリオを許可する必要があります。

以下の式を使用して、分散キャッシュのデータセットサイズの推定を調整できます。

```
Distributed data set size = Data set size * (Number of owners + 1)
```

分散キャッシュでの利用可能なメモリーの計算

分散キャッシュを使用すると、ノードを追加するか、ノードごとに利用可能なメモリーの量を増やすことで、データセットサイズを増やすことができます。

```
Distributed data set size <= Available memory per node * Minimum number of nodes
```

ノードの損失への耐性調整

クラスター内に特定のノード数を配置するように計画する場合でも、すべてのノードがクラスターに常に存在するわけではない点を考慮する必要があります。分散キャッシュは、データが損失することなく、**所有者数 - 1** 分の損失に対応します。そのため、データセットに必要な最小ノード数に加え、その分を余分に割り当てるようにしてください。

```
Planned nodes = Minimum number of nodes + Number of owners - 1

Distributed data set size <= Available memory per node * (Planned nodes - Number of owners + 1)
```

たとえば、サイズが 10KB の 100 万のエントリーを保存して、エントリーごとに 3 つの所有者を設定する計画を予定します。クラスター内の各ノードに 4 GB の RAM を割り当てる予定の場合には、以下の式を使用してデータセットに必要なノード数を判断できます。

```
Data set size = 1_000_000 * 10KB = 10GB
Distributed data set size = (3 + 1) * 10GB = 40GB
40GB <= 4GB * Minimum number of nodes
Minimum number of nodes >= 40GB / 4GB = 10
Planned nodes = 10 + 3 - 1 = 12
```

2.2.1. メモリーのオーバーヘッド

メモリーのオーバーヘッドは、Data Grid がエントリーを保存するために使用する追加のメモリーです。メモリーのオーバーヘッドの概算値は、JMV ヒープメモリーのエントリーごとに 200 バイトまたは、オフヒープメモリーのエントリーごとに 60 バイトです。ただし、Data Grid がエントリーごとに追加するオーバーヘッドは、複数の要因によって異なるため、メモリーのオーバーヘッドを正確に決定

することはできません。たとえば、エビクションでデータをバインドすると、Data Grid は追加のメモリーを使用してエントリーを追跡します。同様に、有効期限を設定すると、タイムスタンプのメタデータが各エントリーに追加されます。

種類に関係なく、メモリーのオーバーヘッドの実際の量を見つけ出す唯一の方法として、JVM ヒープダンプ分析が行われます。当然、JVM ヒープダンプでは、オフヒープメモリーに格納するエントリーの情報はありませんが、メモリーのオーバーヘッドは JVM ヒープメモリーよりも大幅に小さくなります。

追加のメモリー使用量

Data Grid がエントリーごとに課すメモリーオーバーヘッドに加え、リバランスやインデックスなどのプロセスの全体的なメモリー使用量が増える可能性があります。ノードの参加時や退出時にクラスターの操作をリバランスすると、クラスターメンバー間のエントリーを複製している間にデータが損失されないように、一時的に余分な容量が必要になります。

2.2.2. JVM ヒープ領域の割り当て

要件対応に十分なデータストレージ容量を確保できるように、Data Grid のデプロイメントに必要なメモリーの量を決定します。

重要

ガベージコレクション (GC) 時間の設定と組み合わせて大きなメモリーヒープサイズを割り当てると、Data Grid デプロイメントのパフォーマンスに次のような影響を与える可能性があります。

- JVM が 1 つのスレッドしか処理しない場合、GC がスレッドをブロックし、JVM のパフォーマンスを低下させる可能性があります。GC はデプロイ前に動作する場合があります。この非同期動作により、大規模な GC 一時停止が発生する可能性があります。
- CPU リソースが少なく、GC がデプロイと同期して動作する場合、GC はより頻繁に実行する必要があり、デプロイのパフォーマンスが低下する可能性があります。

次の表に、データストレージ用の JVM ヒープ領域を割り当てる 2 つの例を示します。これらの例は、クラスターを安全にデプロイするための想定要件を表しています。

読み取り、書き込み、削除操作などのキャッシュ操作のみ。	データストレージに JVM ヒープ領域の 50% を割り当てる
キャッシュ操作およびクエリーやキャッシュイベントリスナーなどのデータ処理	データストレージに JVM ヒープ領域の 33% を割り当てる

注記

パターンの変更とデータストレージの使用状況に応じて、推奨される安全な見積もりとは異なる割合を JVM ヒープ領域に設定することを検討してください。

Data Grid のデプロイメントを開始する前に、安全な見積もりを設定することを検討してください。デプロイを開始したら、JVM のパフォーマンスとヒープスペースの占有率を確認します。JVM のデータ使用量とスループットが大幅に増加した場合、JVM ヒープ領域の再調整が必要になる場合があります。

安全な見積もりは、次の一般的な操作が JVM 内で実行されているという前提で計算されました。リストは完全なものではなく、追加の操作を実行する目的で、これらの安全な見積もりのいずれかを設定する場合があります。

- Data Grid は、シリアル化された形式のオブジェクトをキーと値のペアに変換します。Data Grid は、ペアをキャッシュと永続ストレージに追加します。
- Data Grid は、リモート接続からクライアントへのキャッシュを暗号化および復号化します。
- Data Grid は、キャッシュの定期的なクエリーを実行してデータを収集します。
- Data Grid は、データを戦略的にセグメントに分割して、状態遷移操作中であってもクラスター間でデータを効率的に分散できるようにします。
- JVM が GC 操作に大量のメモリーを割り当てたため、GC はより頻繁にガベージコレクションを実行します。
- GC は、JVM ヒープスペース内のデータオブジェクトを動的に管理および監視して、未使用のオブジェクトを安全に削除できるようにします。

データストレージに JVM ヒープスペースを割り当てるとき、および Data Grid デプロイメントのメモリーボリュームと CPU 要件を決定するときは、次の要素を考慮してください。

- クラスター化されたキャッシュモード
- セグメント数
  - たとえば、セグメント数が少ないと、サーバーがノード間でデータを分散する方法に影響を与える可能性があります。
- 読み取りまたは書き込み操作。
- リバランス要件。
  - たとえば、状態の転送中に多数のスレッドが並列ですばやく実行される可能性がありますが、各スレッド操作はより多くのメモリーを使用する可能性があります。
- クラスターのスケーリング
- 同期または非同期のレプリケーション

最も主要な Data Grid 操作で、CPU リソースを多く必要とするものには、Pod の再起動後のノードのリバランス、データに対するインデックスクエリーの実行、GC 操作の実行などが挙げられます。

オフヒープストレージ

Data Grid はオブジェクトの JVM ヒープ表現を使用して、キャッシュでの読み取りおよび書き込み操作を処理したり、状態遷移操作などの他の操作を実行します。エントリーをオフヒープメモリーに保存していても、JVM ヒープ領域を Data Grid に常に割り当てる必要があります。

Data Grid がオフヒープストレージで使用する JVM ヒープメモリーの量は、JVM ヒープスペースにデータを格納する場合と比べてはるかに小さくなります。オフヒープストレージの JVM ヒープメモリー要件は、格納されているエントリーの数に対する同時操作の数に比例します。

Data Grid はトポロジーキャッシュを使用して、クライアントにクラスタービューを提供します。

Data Grid クラスターから **OutOfMemoryError** 例外を受け取った場合は、次のオプションを検討してください。

- 状態遷移操作を無効にします。これにより、ノードがクラスターに参加またはクラスターから離脱した場合にデータが失われる可能性があります。
- キーサイズとノードおよびセグメントの数を考慮して、JVM ヒープスペースを再計算します。
- より多くのノードを使用して、クラスターのメモリー消費をより適切に管理します。
- メモリー使用量が少なくて済む可能性があるため、単一ノードを使用します。ただし、クラスターを元のサイズにスケーリングする場合は、影響を考慮してください。

2.3. クラスター化されたキャッシュモード

クラスター化された Data Grid キャッシュを複製または分散として設定できます。

分散キャッシュ

クラスター全体で各エントリーのコピー作成を減らして、容量を最大化します。

レプリケートされたキャッシュ

クラスター内の各ノードにすべてのエントリーのコピーを作成して冗長性を提供します。

読み取り操作 : 書き込み操作

アプリケーションによる書き込み操作を多くするか、読み取り操作を多くするか検討します。通常、分散キャッシュは書き込み操作で最適なパフォーマンスを、レプリケートキャッシュは読み取り操作で最適なパフォーマンスを実現できます。

所有者が 2 名のノード 3 つで設定されるクラスターの分散キャッシュに **k1** を配置するには、Data Grid は **k1** を 2 回書き込みます。レプリケートされたキャッシュで同じ操作を行うと、Data Grid は **k1** を 3 回書き込みます。レプリケートキャッシュへの各書き込みの追加ネットワークトラフィックの量は、クラスター内のノード数と等しくなります。ノードが 10 個あるクラスターのレプリケートキャッシュでは、書き込みのトラフィクが 10 倍になるなどです。クラスタートランスポートのマルチキャストで UDP スタックを使用して、トラフィックを最小限に抑えることができます。

レプリケートキャッシュから **k1** を取得するには、ノードごとに、ローカルに読み取り操作を実行できます。分散キャッシュから **k1** を取得するには、操作を処理するノードはクラスター内の別のノードからキーを取得しないといけない場合があります。これにより、ネットワークホップが追加で発生し、読み取り操作が完了するまで時間が長くなります。

クライアントのインテリジェンスとニアキャッシュ

Data Grid は、一貫性のあるハッシュ技術を使用して、Hot Rod クライアントがトポロジーに対応するようにし、ネットワークホップが追加で発生しないようにすることで、分散キャッシュでも、レプリケートきゃっしゅでも、読み取り操作が同程度のパフォーマンスを出せるようにします。

また、Hot Rod クライアントは、ニアキャッシュ機能を使用して、頻繁にアクセスされるエントリーをローカルメモリーで保持し、読み込みの繰り返しを回避することもできます。

ヒント

分散キャッシュは、ほとんどの Data Grid Server デプロイメントの選択肢として最適です。クラスタースケーリングの弾力性と合わせて、読み込み、書き取り操作でできるだけベストなパフォーマンスを獲得できます。

データの保証

各ノードにすべてのエントリーが含まれるため、レプリケートキャッシュは分散キャッシュよりも、データが損失されないように保護されます。3 つのノードで設定されるクラスターでは、2 つのノードがクラッシュでき、複製キャッシュからデータが失われることはありません。

同じシナリオでは、所有者が 2 人の分散キャッシュでは、データが損失します。分散キャッシュによるデータの損失を回避するには、**owners** 属性を宣言するか、**numOwners()** メソッドで各エントリーの所有者をさらに設定し、クラスター全体でのレプリカ数を増してください。

ノードの障害発生時の操作のリバランス

ノードの障害後に操作をリバランスすると、パフォーマンスや容量に影響を与える可能性があります。ノードがクラスターから退出すると、Data Grid は残りのメンバー間でキャッシュエントリーを複製して、設定された所有者数を復元します。このリバランス操作は一時的なものですが、クラスターのトラフィックが増加すると、パフォーマンスに悪影響を及ぼします。ノードの退出が増えると、パフォーマンスがさらに低下します。多数のノードが退出すると、クラスター内に残っているノードの容量が足りなくなり、メモリーに全データを保持できなくなる可能性があります。

クラスターのスケーリング

Data Grid クラスターは、ワークロードのニーズに合わせて水平的にスケーリングし、CPU やメモリーなどのコンピュートリソースをより効率的に使用します。この弾力性を最大限に活用するには、ノード数の増減でスケーリングすることでどの程度キャッシュの容量に影響があるかを検討する必要があります。

レプリケートキャッシュの場合は、ノードがクラスターに参加するたびに、データセットの完全なコピーを受け取ります。各ノードに全エントリーを複製すると、ノードの参加にかかる時間が長くなり、全体的な容量に制限が課せられます。レプリケートキャッシュは、ホストで利用可能なメモリー量を超えることはできません。たとえば、データセットのサイズが 10 GB の場合、各ノードに少なくとも 10 GB のメモリーが必要です。

分散キャッシュの場合は、クラスターの各メンバーがデータのサブセットのみを格納するため、ノードを追加すると容量も増加します。10 GB のデータを保存するには、所有者数が 2 つで、メモリーのオーバーヘッドを考慮しない場合には、利用可能なメモリーが 5 GB のノードを 8 個指定すると良いでしょう。追加のノードがクラスターに参加すると、分散キャッシュの容量が 5GB 増加します。

分散キャッシュの容量は、基盤となるホストで利用可能なメモリー量により、制限されることはありません。

同期または非同期のレプリケーション

Data Grid は、プライマリーの所有者がレプリケーション要求をバックアップノードに送信すると、同期または非同期に通信できます。

レプリケーションモード	パフォーマンスへの影響
同期	同期レプリケーションは、データの整合性を保つのに役立ちますが、クラスタートラフィックに対するレイテンシーが増え、キャッシュの書き込みのスループットが減少します。
非同期	非同期レプリケーションでは、レイテンシーが短くなり、書き込み操作の速度が早くなりますが、データの不整合が発生するため、データ損失に対する保証が少なくなります。

同期レプリケーションでは、Data Grid は、バックアップノードでレプリケーション要求の完了時に元のノードに通知します。Data Grid は、クラスタートポロジーの変更が原因でレプリケーション要求に失敗すると、操作を再試行します。他のエラーが原因でレプリケーション要求に失敗すると、Data Grid はクライアントアプリケーションの例外を出力します。

非同期のレプリケーションでは、Data Grid はレプリケーション要求の確認は行いません。これは、すべての要求が正常に実行されるのと同じ効果を持ちます。ただし、Data Grid クラスターでは、プライマリーの所有者のエントリーが正しく、Data Grid は今後のある時点でバックアップノードに複製を作成します。プライマリー所有者がクラッシュすると、バックアップノードにエントリーのコピーがない場合や、以前のコピーが含まれる可能性があります。

クラストポロジーが変更されたことが原因で、非同期のレプリケーションでデータの不整合が生じる可能性もあります。たとえば、プライマリー所有者が複数指定されている Data Grid クラスターについて考えてみましょう。ネットワークエラーまたはその他の問題により、1 つ以上のプライマリー所有者が予期せずにクラスターから退出するため、Data Grid により、どのノードがどのセグメントのプライマリー所有者であるかが更新されます。これが発生すると、一部のノードで以前のクラストポロジーを使用し、他のノードで更新されたトポロジーを使用することが理論的に可能です。非同期通信では、Data Grid が以前のトポロジーからのレプリケーション要求を処理し、書き込み操作から以前の値を適用する時間が短縮される可能性があります。ただし、Data Grid はノードクラッシュを検出し、このシナリオから多くの書き込み操作に影響が及ばないように、素早くクラスターのトポロジーの変更を更新します。

非同期レプリケーションでは、1 回にノードが処理できるバックアップの書き込み数を、考えられる送信者数だけに限定するため、非同期のレプリケーションを使用しても、書き込みのスループットは保証されません。同期レプリケーションにより、ノードはより多くの受信書き込み操作を同時に処理できるようになり、特定の設定では個別の操作の完了に時間がかかることが考慮され、合計スループットが多くなっています。

ノードがエントリー複製の要求を複数送信すると、JGroups はクラスターの残りのノードに一度に 1 つずつメッセージを送信するので、送信元のノード別の複製要求は 1 つだけ作成されることになります。つまり、Data Grid ノードは他の書き込み操作と並行して処理でき、クラスター内の他のノードから書き込みを行うことができます。

Data Grid は、クラストランスポート層で JGroups フロー制御プロトコルを使用して、バックアップノードへのレプリケーション要求を処理します。未確認のレプリケーション要求の数が、**max_credits** 属性 (デフォルトでは 4MB) で設定されたフロー制御のしきい値を超えると、送信元ノードで書き込み操作がブロックされます。これは、同期および非同期レプリケーションの両方に適用されます。

セグメント数

Data Grid は、データをセグメントに分割して、データをクラスター全体に均等に分散します。セグメントを均等分散すると、個別のノードのオーバーロードが回避され、クラスターのリバランス操作をより効率化します。

Data Grid はデフォルトで、クラスターごとに 256 ハッシュ領域セグメントを作成します。クラスターごとに最大 20 個のノードで設定されるデプロイメントの場合には、このセグメント数が理想的であるため、変更しないでください。

クラスターごとに 20 を超えるノードで設定されるデプロイメントの場合には、セグメント数を増やすと、データの詳細レベルが上がるので、Data Grid はクラスター全体に効率的に分散できるようにします。設定するセグメント数を算出するには、以下の式を使用します。

```
Number of segments = 20 * Number of nodes
```

たとえば、クラスターが 30 個ある場合は、600 セグメントを設定する必要があります。大規模なクラスターのセグメントを増やすことは、一般的には適切ですが、この公式でデプロイメントに適したおおよその数がわかります。

Data Grid が作成するセグメント数を変更するには、クラスターを完全に再起動する必要があります。永続ストレージを使用する場合は、キャッシュストアの実装によっては、**StoreMigrator** ユーティリティーを使用してセグメント数を変更する必要もあります。

セグメント数を変更すると、データが破損する可能性があるため、ベンチマーキングおよびパフォーマンスの監視で収集したメトリックをもとに、注意して変更する必要があります。

注記

Data Grid は、メモリーに格納するデータを常にセグメント化します。キャッシュストアの設定時には、Data Grid では永続ストレージでデータのセグメント化が行われるわけではありません。

キャッシュストアの実装に依存しますが、可能な場合にはキャッシュストアのセグメント化を有効にしてください。セグメント化されたキャッシュストアは、永続ストレージのデータを反復処理する時に、Data Grid のパフォーマンスを向上させます。たとえば、RocksDB と JDBC 文字列ベースのキャッシュストアを使用すると、セグメント化により、Data Grid がデータベースから取得する必要のあるオブジェクトの数が減ります。

2.4. 古いデータを管理するストラテジー

Data Grid がデータのプライマリーソースではない場合に、性質的に埋め込みキャッシュとリモートキャッシュは古くなります。Data Grid デプロイメントのプランニング、ベンチマーク、およびチューニング時に、アプリケーションに適したキャッシュの有効期限を選択します。

利用可能な RAM を最大限にできるようにレベルを選択し、キャッシュミスを回避します。Data Grid にエントリーがメモリーにない場合は、アプリケーションが読み取りおよび書き込み要求を送信する時にプライマリーストアに送信されます。

キャッシュミスにより、読み取りおよび書き込みのレイテンシーが増えますが、多くの場合、プライマリーストアへの呼び出しは Data Grid のパフォーマンス低下よりもコストが高くなります。たとえば、一例として、リレーショナルデータベース管理システム (RDBMS) を Data Grid クラスターにオフロードします。このように Data Grid をデプロイすると、従来のデータベースの運用におけるコストが大幅に削減されるため、キャッシュに古くなったエントリーを許容するのは合理的です。

Data Grid では、エントリーに対してアイドルおよびライフスパンの値を最大で設定し、キャッシュの許容できる有効期限を保ちます。

有効期限

Data Grid がエントリーをキャッシュに保存する期間や、クラスター全体で有効な期間を制御します。

有効期限の値が大きくなると、エントリーがメモリーに保存される期間が長くなり、読み込み操作で古い値が渡される可能性が高くなります。有効期限の値が小さいほどキャッシュに含まれる古い値が少なくなりますが、キャッシュミスが発生する可能性が高くなります。

有効期限を指定するため、Data Grid は既存のスレッドプールからリーパーを作成します。スレッドでのパフォーマンスに関する考慮事項として主に、次に有効期限の実行を行うまでの間隔を適切に設定することが挙げられます。間隔が短いほど有効期限が頻繁に実行されますが、より多くのスレッドを使用します。

さらに、アイドル状態の最大有効期限を使用して、Data Grid がクラスター全体でタイムスタンプを更新する方法を制御できます。Data Grid は touch コマンドを送信して、ノード全体で最大アイドル有効期限を同期的、または非同期的に連携します。同期レプリケーションでは、整合性または速度のどちらを優先するかに応じて、sync または async の touch コマンドのいずれかを選択できます。

2.5. エビクションによる JVM メモリー管理

RAM はコストの高いリソースで、通常、利用できる量に限りがあります。Data Grid を使用すると、メモリーからエントリーを削除することで、頻繁に使用されるデータを優先するように、メモリー使用量を管理できます。

エビクション

Data Grid がメモリー内に維持し、各ノードに対して有効になるデータ量を制御します。

エビクションは、以下で Data Grid のキャッシュをバインドします。

- エントリーの合計数 (最大数)。
- JVM メモリーサイズ (最大サイズ)

重要

Data Grid は、ノードごとにエントリーをエビクトします。すべてのノードが同じエントリーをエビクトしないため、永続ストレージでエビクションを使用して、データの不整合を回避する必要があります。

エビクションからパフォーマンスへの影響は、キャッシュのサイズが設定したしきい値に到達すると Data Grid が算出する必要のある追加の処理内容からきています。

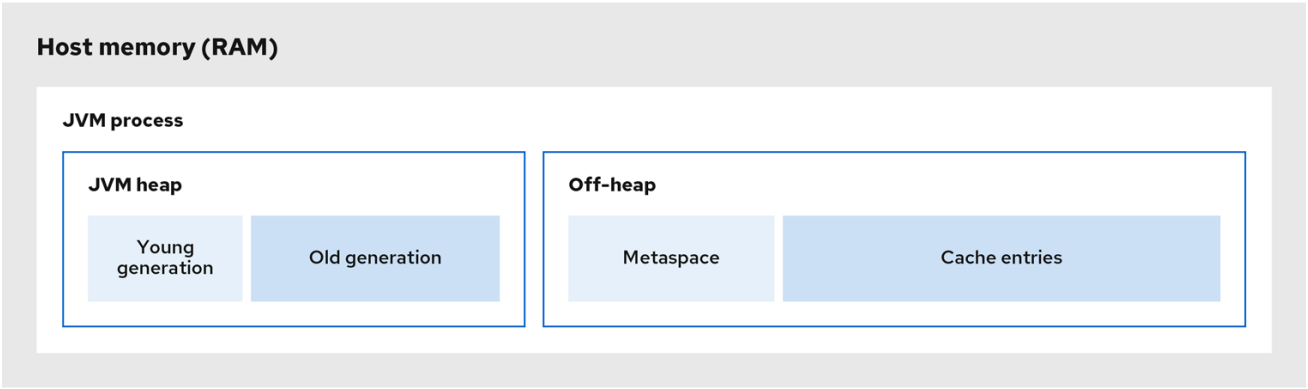
エビクションは読み取り操作の速度を低下させることもできます。たとえば、読み取り操作がキャッシュストアからエントリーを取得する場合に、Data Grid はそのエントリーをメモリーに送り、別のエントリーをエビクトします。このエビクションプロセスでは、パッシベーションを使用する場合に、新たにエビクトされたエントリーをキャッシュストアに書き込むことができます。これが発生すると、読み取り操作はエビクションプロセスが完了するまで値を返しません。

2.6. JVM ヒープおよびオフヒープメモリー

Data Grid は、キャッシュエントリーを、デフォルトで JVM ヒープメモリーに保存します。Data Grid は、オフヒープストレージを使用するように設定できます。つまり、管理された JVM メモリー領域外のネイティブメモリーが、データで占有されます。

以下の図は、Data Grid が実行している JVM プロセスのメモリー領域を簡略して示しています。

図2.1 JVM メモリー領域



184_Data_Grid_0921

JVM ヒープメモリー

ヒープは、参照される Java オブジェクトや他のアプリケーションデータをメモリーに維持するのに役立つ新しい世代と古い世代に分けられます。GC プロセスは、到達不能オブジェクトから領域を回収し、新しい生成メモリープールでより頻繁に実行します。

Data Grid がキャッシュエントリーを JVM ヒープメモリーに保存すると、キャッシュへのデータ追加を開始するため、GC の実行が完了するまで時間がかかる場合があります。GC は集中的なプロセスであるため、実行が長く頻繁になると、アプリケーションのパフォーマンスが低下する可能性があります。

オフヒープメモリー
オフヒープメモリーは、JVM メモリー管理以外のネイティブで利用可能なシステムメモリーです。**JVM メモリーの領域** の図には、クラスメタデータを保持し、ネイティブメモリーから割り当てられる **メタスペース** メモリープールが表示されます。この図は、Data Grid キャッシュエントリーを保持するネイティブメモリーのセクションも含まれています。

オフヒープメモリー

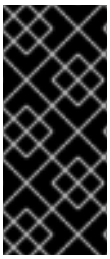
- エントリーごとに少ないメモリーを使用します。
- Garbage Collector (GC) の実行を回避するために、JVM 全体のパフォーマンスを改善します。

ただし、JVM ヒープダンプはオフヒープメモリーに保存されているエントリーを表示しない点が短所です。

2.6.1. オフヒープデータストレージ

オフヒープキャッシュにエントリーを追加すると、Data Grid はネイティブメモリーをデータに動的に割り当てます。

Data Grid は、各キーのシリアル化された **byte []** を、標準の Java **HashMap** と同様のバケットにハッシュ値を持ちます。バケットには、Data Grid がオフヒープメモリーに保存するエントリーの検索に使用するアドレスポインターが含まれます。



重要

Data Grid はキャッシュエントリーをネイティブメモリーに保存する場合でも、ランタイム操作にはこれらのオブジェクトの JVM ヒープ表現が必要です。たとえば、**cache.get()** 操作は、返される前にオブジェクトをヒープメモリーに読み取ります。同様に、状態遷移操作は、オブジェクトのサブセットが実行している間は、それらをヒープメモリーに保持します。

オブジェクトの等価性

Data Grid は、オブジェクトインスタンスではなく、各オブジェクトのシリアライズされた byte[] 表現を使用して、オフヒープストレージで Java オブジェクトの等価性を決定します。

データの整合性

Data Grid は、ロックの配列を使用して、オフヒープアドレス空間を保護します。ロックの数は、コア数に 2 倍になり、その後に最も近い 2 の累乗に丸められます。これにより、書き込み操作が読み取り操作をブロックしないように、**ReadWriteLock** インスタンスの配分も存在します。

2.7. 永続ストレージ

永続データソースと対話するように Data Grid を設定すると、パフォーマンスに大幅に影響を及ぼします。このパフォーマンスのペナルティーは、従来型のデータソースが本質的にインメモリーキャッシュよりも遅いという事実から来ています。読み取り操作と書き込み操作はいつでも、呼び出しが JVM 外

で行われると時間がかかります。キャッシュストアの使用方法に応じて、Data Grid パフォーマンスは、永続ストレージのデータにアクセスする場合と比べ、インメモリーデータがパフォーマンスを向上するので、パフォーマンスが低下されるのを抑えることができます。

永続ストレージで Data Grid デプロイメントを設定すると、クラスターの正常なシャットダウンの状態を保持できるなど、他の利点も得られます。また、キャッシュから永続ストレージにデータをオーバーフローし、メモリーだけで利用できる容量以上のキャパシティーを獲得できます。たとえば、メモリーに 200 万程度だけを保持しつつ、合計 1000 万個のエントリーを確保できます。

Data Grid は、ライトスルーモードまたはライトビハインドモードで、キー/値のペアをキャッシュと永続ストレージに追加します。これらの書き込みモードはパフォーマンスに別の影響を与えるため、Data Grid デプロイメントの計画時に考慮する必要があります。

書き込みモード	パフォーマンスへの影響
ライトスルー	Data Grid は、キャッシュと永続ストレージに同時にデータを書き込むため、一貫性が向上してノード障害などが原因のデータ損失を回避します。 ライトスルーモードの欠点は、同期書き込みが原因でレイテンシーが増加してスループットを減少する点です。**Cache.put()** 呼び出しにより、永続ストレージへの書き込みが完了するまでアプリケーションスレッドが待機します。
ライトビハインド	Data Grid はデータを同期的に書き込みますが、変更をキューに追加して永続ストレージへの書き込みが非同期に行われるようにします。これにより、一貫性は低下しますが、書き込み操作のレイテンシーが短縮されます。 キャッシュストアが書き込み操作の数を処理できない場合には、Data Grid は、保留中の書き込み操作の数が、ライトスルーと同じ方法で、設定した変更キューサイズを下回るまで、新しい書き込みを遅延させます。ストアの速度は十分購読ですが、キャッシュの書き込みが急増し、レイテンシーが急激に伸びた場合には、変更キューサーズを増やして、急増に対応してレイテンシーを減らすことができます。

パッシベーション

パッシベーションを有効にすると、Data Grid がエントリーをメモリーからエビクトする場合にのみ永続ストレージにエントリーを書き込むように設定されます。パッシベーションは、アクティベーションの意も含まれます。キーの読み取りまたは書き込みを実行すると、その鍵がメモリーに戻されて永続ストレージから削除されます。アクティベーション中に永続ストレージからキーを削除しても、読み取り操作や書き込み操作はブロックされませんが、外部ストアへの負荷が増大します。

パッシベーションおよびアクティベーションは、Data Grid が、キャッシュ内の特定のエントリーに対して、永続ストレージへの複数の呼び出しを実行する可能性があります。たとえば、エントリーがメモリーで利用できない場合には、Data Grid はエントリーをメモリーに戻して (読み取り操作 1 回)、永続ストレージから削除します。また、キャッシュのサイズ制限に達すると、Data Grid は別の書き込み操作を実行して、新たにエビクトされたエントリーを渡します。

データによるキャッシュの事前ロード

Data Grid クラスターのパフォーマンスに影響を与える可能性のある永続ストレージの他の側面として、キャッシュの事前読み込みが挙げられます。この機能により、Data Grid クラスターの起動時にキャッシュにデータを設定し、これらのキャッシュはウォームな状態であるため、読み取りおよび書き込みをより簡単に処理できます。キャッシュが事前読み込みされている場合には、Data Grid クラスターの起動時間が遅くなり、永続ストレージのデータの量が利用可能な RAM の容量よりも大きい場合に、メモリー不足の例外が生じる可能性があります。

2.8. クラスターセキュリティー

データを保護すると共に、ネットワーク侵入を防ぐことは、デプロイメント計画で最も重要な要素です。機密な顧客情報が公開インターネットに漏洩したり、データが侵害され、ハッカーが機密情報を一般に公開できるようになるので、ビジネスの評判に大打撃を与えます。

これを念頭に、ユーザーを認証し、ネットワーク通信を暗号化するには、強固なセキュリティーストラテジーが必要になります。では、Data Grid デプロイメントのパフォーマンスに対する代償は何ですか?計画時のこれらの考慮事項はどのように調整すればいいですか

認証
ユーザー認証情報の検証時のパフォーマンスに対する影響は、メカニズムとプロトコルによって異なります。Data Grid は、Hot Rod 経由でユーザーごとに認証情報を 1 回検証し、HTTP 経由で全要求を場合によっては検証する場合があります。

表2.1 認証メカニズム

SASL メカニズム	HTTP メカニズム	パフォーマンスへの影響
PLAIN	**BASIC**	**PLAIN** および **BASIC** は最も高速な認証メカニズムですが、セキュリティーも最も低くなります。TLS/SSL 暗号化と組み合わせた **PLAIN** または **BASIC** のみを使用する必要があります。
DIGEST および **SCRAM**	**DIGEST**	Hot Rod および HTTP 要求の両方の場合には、**DIGEST** スキーム は MD5 ハッシュアルゴリズムを使用して認証情報をハッシュ化し、プレーンテキストで送信されないようにします。TLS/SSL 暗号化を有効にしない場合には、**DIGEST** は暗号化による **PLAIN** または **BASIC** よりも全体的なリソース集約型ですが、**DIGEST** は中間者 (MITM) 攻撃やその他の侵入に対して脆弱であるため、安全ではありません。 Hot Rod エンドポイントの場合には、**SCRAM** スキームは**DIGEST** と似ていますが、保護レベルが高く、セキュリティーが強化されていますが 、完了までに追加の処理が必要になります。
GSSAPI / **GS2-KRB5**	**SPNEGO**	Kerberos サーバー、キー配布センター (KDC) は認証を処理し、ユーザーにトークンを発行します。Data Grid のパフォーマンスは、別のシステムがユーザー認証操作を処理するという利点が得られます。ただし、これらのメカニズムにより、KDC サービス自体のパフォーマンスによっては、ネットワークのボトルネックが生じる可能性があります。

SASL メカニズム	HTTP メカニズム	パフォーマンスへの影響
OAUTHBEARER	**BEARER_TOKEN**	Data Grid ユーザーに一時的なアクセストークンを発行するための OAuth 標準を実装するフェデレーションされたアイデンティティープロバイダー。ユーザーは、Data Grid に直接認証するのではなく、アイデンティティーサービスで認証し、アクセストークンを要求ヘッダーとして渡します。認証を直接処理する場合と比較すると、Data Grid がユーザーのアクセストークンを検証する場合のパフォーマンスへの影響が少なくなります。KDC と同様に、実際のパフォーマンスの影響は、アイデンティティープロバイダー自体の QoS(Quality of Service) によって異なります。
EXTERNAL	**CLIENT_CERT**	Data Grid Server にトラストストアを提供し、クライアントが信頼ストアに対して提示する証明書を比較することで、受信接続を認証できるようにします。 トラストストアに、通常は認証局 (CA) である署名証明書のみが含まれる場合 (通常は証明局 (CA))、その CA が署名した証明書を提示するクライアントは Data Grid に接続できます。これによりセキュリティーが低下し、MITM 攻撃に対して脆弱になりますが、クライアントごとに公開証明書を認証するよりも高速です。 トラストストアに、署名証明書に加えてすべてのクライアント証明書が含まれる場合には、トラストストアに存在する署名済み証明書を提示するクライアントのみが Data Grid に接続できます。この場合、Data Grid は、証明書が署名されていることを検証せずに、クライアントが提示する証明書の共通の Common Name (CN) をトラストストアと比較し、より多くのオーバーヘッドを追加します。

暗号化

クラストートランスポートを暗号化すると、ノード間でデータを渡して、MITM 攻撃から Data Grid デプロイメントを保護するので、データのセキュリティーを確保できます。ノードは、クラスターに参加する際に TLS/SSL ハンドシェイクを実行し、パフォーマンスが大幅に低下し、追加のラウンドトリップのレイテンシーが増加します。ただし、各ノードが接続を確立すると、接続がアイドル状態にならないことを想定して、そのままずっと接続を確保します。

リモートキャッシュの場合、Data Grid Server はクライアントとのネットワーク通信を暗号化することもできます。パフォーマンス上の観点では、クライアントとリモートキャッシュ間の TLS/SSL 接続の影響は同じです。セキュアな接続の交渉に時間がかかり、追加の作業が必要になりますが、接続が確立されると、暗号化からのレイテンシーが原因で Data Grid のパフォーマンス影響を与えることはありません。

TLSv1.3 を使用する以外に、暗号化を使用した時のパフォーマンスの低下を緩和する方法として唯一、Data Grid を実行する JVM を設定することができます。たとえば、標準の Java 暗号化の代わりに OpenSSL ライブラリーを使用する場合は、結果として最大 20% 早くなり、より効率的に処理できま

す。

認可
ロールベースアクセス制御 (RBAC) により、データに対する操作を制限でき、追加のセキュリティーをデプロイメントに提供します。RBAC は、Data Grid クラスター全体に分散されたデータに、最小限のユーザーアクセス権限を指定するポリシーを実装するのに最適な方法です。Data Grid ユーザーは、キャッシュからデータの読み取り、作成、変更、または削除を行うのに十分なレベルの権限が必要です。

別のセキュリティーレイヤーを追加してデータを保護すると、常にパフォーマンスが低下します。Data Grid は、ユーザーによるデータの操作を許可する前にアクセス制御リスト (ACL) で各操作を検証するので、認可の操作ではレイテンシーが長くなります。ただし、認可によるパフォーマンスへの影響は、暗号化よりもはるかに低いため、通常、利点で欠点が補填されます。

2.9. クライアントリスナー

クライアントリスナーは、Data Grid クラスターでデータを追加、削除、または変更されるたびに通知を出します。

たとえば、以下の実装は、指定の場所で温度が変わるたびにイベントをトリガーします。

```
@ClientListener
public class TemperatureChangesListener {
  private String location;

  TemperatureChangesListener(String location) {
    this.location = location;
  }

  @ClientCacheEntryCreated
  public void created(ClientCacheEntryCreatedEvent event) {
    if(event.getKey().equals(location)) {
      cache.getAsync(location)
           .whenComplete((temperature, ex) ->
                System.out.printf(">> Location %s Temperature %s", location, temperature));
    }
  }
}
```

Data Grid クラスターにリスナーを追加すると、デプロイメントのパフォーマンスの考慮事項が追加されます。

埋め込みキャッシュの場合には、リスナーは Data Grid と同じ CPU コアを使用します。多くのイベントを受信し、大量の CPU を使用してこれらのイベントを処理するリスナーは、Data Grid が使用できる CPU を減らし、その他の操作の速度を低下させます。

リモートキャッシュの場合には、Data Grid Server は内部プロセスを使用してクライアント通知をトリガーします。Data Grid Server は、イベントをプライマリー所有者ノードからクライアントに送信する前に、リスナーの登録先のノードに送信します。Data Grid Server には、クライアントリスナープロセスイベントの速度が遅い場合にキャッシュへの書き込み操作を遅延させるバックプレシャーメカニズムも含まれています。

リスナーイベントのフィルタリング

すべての書き込み操作でリスナーが呼び出されると、Data Grid は多数のイベントを生成し、クラスター内にネットワークトラフィックと外部クライアントの両方を作成できます。これらはすべて、各リ

スナーに登録されているクライアントの数、トリガーするイベントのタイプ、および Data Grid クラスターのデータ変更によって異なります。

リモートキャッシュが設定された例として、イベントを 10 個生成するリスナーで 10 台のクライアントを登録すると、Data Grid Server はネットワーク全体で 100 個のイベントを送信します。

Data Grid Server にカスタムフィルターを指定して、クライアントへのトラフィックを減らすことができます。フィルターを使用すると、Data Grid Server が最初にイベントを処理して、イベントをクライアントに転送するかどうかを判断できます。

継続クエリーおよびリスナー

継続クエリーを行うことで、一致するエントリーのイベントを受信し、クライアントリスナーのデプロイや、リスナーイベントのフィルタリングの代わりとなる手段を提供します。当然、クエリーには、追加の処理コストがかかる点を考慮する必要がありますが、クライアントリスナーではなく、継続クエリーを使用してキャッシュのインデックス化、クエリーの実行を行っている場合には、価値がある場合があります。

2.10. キャッシュのインデックス作成とクエリー

Data Grid キャッシュをクエリーすると、データを分析してフィルタリングし、リアルタイムで知見を得ることができます。たとえば、プレイヤー同士が競争して点数を獲得するオンラインゲームを思い浮かべてください。上位 10 位のプレイヤーを表示するリーダーボードを実装する場合は、以下のように、ある時点で最高得点を獲得したプレイヤーを特定して、最大 10 位までの結果に制限するといったクエリーを作成できます。

```
QueryFactory queryFactory = Search.getQueryFactory(playersScores);
Query topTenQuery = queryFactory
  .create("from com.redhat.PlayerScore ORDER BY p.score DESC, p.timestamp ASC")
  .maxResults(10);
List<PlayerScore> topTen = topTenQuery.execute().list();
```

前述の例では、何百万もあるキャッシュエントリーから基準とあったエントリーを 10 件特定できるので、クエリーを使用する利点がわかります。

ただし、パフォーマンスへの影響の面では、インデックス操作とクエリー操作に関するトレードオフを考慮する必要があります。キャッシュをインデックス化するように Data Grid を設定すると、クエリーがはるかに速くなります。インデックスなしでは、クエリーがキャッシュ内のすべてのデータをチェックする必要があるので、データの種類と量に応じて、結果が遅くなります。

インデックス化が有効な場合に書き込みのパフォーマンスがある程度、低下します。インデックスする内容を理解し、慎重にプランニングすることで、最も深刻な影響は回避できます。

最も効果的な方法として、必要なフィールドだけをインデックス化するように Data Grid を設定できます。Plain Old Java Objects(POJO) を保存するか、Protobuf スキーマを使用するかに関係なく、アノテーションをつけるフィールドが多いと、Data Grid がインデックス構築にかかる時間が長くなります。フィールドが 5 つある POJO があるものの、この内 2 つのフィールドだけをクエリーする必要がある場合は、必要のない 3 つのフィールドをインデックス化しないように Data Grid を設定してください。

Data Grid には、インデックス操作を調整するオプションが複数あります。たとえば、Data Grid はデータとは違う形式でインデックスを保存し、メモリーではなく、ディスクにインデックスを保存します。Data Grid は、エントリーが追加、変更、削除されるたびに、インデックスライターを使用してキャッシュとインデックスを同期させます。インデックスを有効にした後、書き込みが遅くなり、インデックスが原因でパフォーマンスが低下したと考えられる場合は、ディスクに書き込む前に、インデックスをメモリーバッファーに長く保持することができます。その結果、インデックス処理が高速化され、書き

込みスループットの低下を抑えることができますが、メモリーが多く消費されます。ただし、ほとんどのデプロイメントでは、デフォルトのインデックス設定が適しており、書き込みがあまり遅くならないようになっています。

たとえば、書き込みの多いキャッシュでは、頻繁にクエリーを実行する必要がなく、ミリ秒単位の結果が必要ない場合など、キャッシュにインデックスを作成しないほうが適しているシナリオもあります。すべて実現する内容により、異なります。クエリーが速くなると、読み込みも速くなりますが、インデックス化すると代わりに書き込みが遅くなります。

maxResults と **hit-count-accuracy** の値を適切に設定することで、インデックス付きクエリーのパフォーマンスを向上させることができます。

関連情報

- Data Grid キャッシュのクエリー

2.10.1. 継続クエリーと Data Grid のパフォーマンス

継続的なクエリーは、アプリケーションに更新の定数ストリームを提供しており、大量のイベントを生成できます。Data Grid は、生成する各イベントにメモリーを一時的に割り当てます。これにより、メモリー不足が発生し、特にリモートキャッシュに対して **OutOfMemoryError** 例外が発生する可能性があります。このため、パフォーマンスへの影響を回避するために、継続的なクエリーを慎重に設計する必要があります。

Data Grid は、継続的なクエリーのスコープを必要な情報の最小量に制限することを強く推奨します。これを実行するには、プロジェクション および述語を使用できます。たとえば、以下のステートメントでは、エントリー全体ではなく基準に一致するフィールドのサブセットのみに関する結果を表示します。

```
SELECT field1, field2 FROM Entity WHERE x AND y
```

また、各 **ContinuousQueryListener** は、ブロッキングスレッドを使用せずに受信したすべてのイベントを迅速に処理できるようにすることが重要です。これを実行するには、イベントを不必要に生成するキャッシュ操作を回避する必要があります。

2.11. データの整合性

分散システム上に存在するデータは、一時的なネットワークの停止やシステム障害、あるいは単純なヒューマンエラーによって生じるエラーなどの影響を受けやすくなっています。このような外部要素については制御できませんが、データの品質に深刻な影響を与える可能性があります。データ破損の影響は、顧客満足度の低下、サービスが利用できなくなるほどのコストのかかるシステムの再構築など多岐にわたります。

Data Grid は ACID(atomic, consistent, isolated, durable) トランザクションを実行すると、キャッシュの状態が一貫していることを確認できます。

トランザクションとは、Data Grid が 1 つの操作として処理する一連の操作のことです。トランザクションのすべての書き込み操作が正常に完了するか、すべてに失敗します。これにより、トランザクションでは、一貫した方法でキャッシュの状態を変更して、読み取りおよび書き込みの履歴を提供するか、キャッシュの状態を全く変更しません。

トランザクションを有効にした時のパフォーマンス上の懸念は主に、データセットの一貫性を高めることと、書き込みスループットを低下させるレイテンシーを増やすこと、この 2 点でバランスを取ることです。

トランザクションを使用した書き込みロック

誤ったロックモードを設定すると、トランザクションのパフォーマンスが低下する可能性があります。Data Grid デプロイメントで、キー間の競合の割合が高いか、引くかにより、適切なロックモードが異なります。

2 つ以上のトランザクションが同じキーに同時に書き込むことがないような、競合率が低いワークロードの場合には、楽観的ロックが最適なパフォーマンスを発揮します。

Data Grid は、トランザクションをコミットする前にキーで書き込みロックを取得します。キーの競合がある場合、ロックの取得にかかる時間が原因でコミットを遅延させる可能性があります。さらに、Data Grid が競合する書き込みを検出すると、トランザクションをロールバックし、アプリケーションによる再試行が必要になり、レイテンシーが長くなります。

競合率の高いワークロードの場合、悲観ロックが最適なパフォーマンスを発揮します。

Data Grid は、アプリケーションがキーにアクセスするときにキーに対する書き込みロックを取得し、他のトランザクションでキーを変更できないようにします。キーはすでにロックされているため、トランザクションのコミットは 1 つのフェーズで完了します。複数のキートランザクションを使用する悲観ロックにより、Data Grid のキーのロック時間が長くなり、書き込みスループットが低下する可能性があります。

読み取りの分離

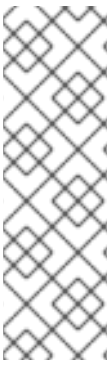
分離レベルは、**REPEATABLE_READ** が指定された楽観的ロックを除き、Data Grid パフォーマンスに関する考慮事項には影響しません。この組み合わせにより、Data Grid は書き込みスキューをチェックして競合を検出するので、トランザクションのコミットフェーズが長くなる可能性があります。また、Data Grid はバージョンメタデータを使用して競合する書き込み操作を検出しますが、これはエントリーごとのメモリー量を増やし、クラスターのネットワークトラフィックを追加で生成する可能性があります。

トランザクションリカバリーおよびパーティション処理

パーティションやその他の問題によりネットワークが不安定になった場合には、Data Grid はトランザクションを in-doubt (疑わしい) とマークできます。このような場合には、Data Grid は、ネットワークの安定化とクラスターが正常な稼働状態に戻されるまで、取得した書き込みロックを保持します。システム管理者が in-doubt (疑わしい) 状態トランザクションを手動で完了させる必要がある場合があります。

2.12. ネットワークパーティションおよび DEGRADED モードのクラスター

Data Grid クラスターでは、クラスター内のノードのサブセットが相互に分離され、ノード間の通信が解除されるスプリットブレインが発生する可能性があります。これが発生すると、少数派のパーティションに含まれる Data Grid キャッシュは **DEGRADED** モードになり、多数派のパーティションのキャッシュは利用可能な状態を維持します。



注記

ネットワークパーティションが発生する最も一般的な原因は、ガベージコレクション (GC) の一時停止です。GC が一時停止になってノードが応答しなくなると、Data Grid クラスターはスプリットブレイン状態のネットワークで動作を開始する可能性があります。

ネットワークパーティションに対応する代わりに、JVM ヒープの使用量を制御して、OpenJDK の Shenandoah など、より新しい一時停止の少ない GC 実装を使用することで、GC 一時停止を回避する必要があります。

CAP 定理およびパーティション処理のストラテジー

CAP 定理は、Data Grid などの分散型のキー/値データストアの制限を表します。ネットワークパーティションイベントが発生した場合は、Data Grid がパーティションを修復し、競合するエントリーを解決する間、整合性と可用性のどちらかを選択する必要があります。

可用性

読み取りおよび書き込みの操作を許可します。

一貫性

読み取りおよび書き込みの操作を拒否します。

また、Data Grid は、クラスターを統合しなおしている間は、読み取りだけが可能です。このストラテジーは、アプリケーションが (古くなった) データにアクセスできるようにして可用性を、エントリーへの書き込みを拒否することで一貫性を保つ、よりバランスの取れたオプションです。

パーティションの削除

クラスターを元のように結合して通常の操作に戻るプロセスの一部として、Data Grid はマージポリシーに従って競合するエントリーを解決します。

デフォルトでは、Data Grid はマージ時に競合を解決しようとしないので、クラスターは、すぐに正常な状態に戻り、通常のクラスターのリバランス以上にパフォーマンスのペナルティーは発生しません。ただし、この場合は、キャッシュ内のデータに一貫性がなくなる可能性が高くなります。

マージポリシーを設定すると、Data Grid がパーティションを修復するのに時間がかかります。マージポリシーを設定すると、Data Grid は各キャッシュからエントリーのすべてのバージョンを取得し、以下のように競合を解決します。

PREFERRED_ALWAYS	Data Grid は、クラスターの過半数のノードに存在する値を見つけ、これを適用するので、古くなった値をリストアできます。
PREFERRED_NON_NULL	Data Grid は、クラスター上で見つかった最初の null 以外の値を適用するので、古くなった値をリストアできます。
REMOVE_ALL	Data Grid は、値が競合するエントリーを削除します。

2.12.1. ガベージコレクションとパーティションの処理

ガベージコレクション (GC) の時間が長くなると、Data Grid がネットワークパーティションの検出に要する時間が長くなる可能性があります。場合によっては、GC が原因で、Data Grid が分割を検出する最大時間を超過する可能性があります。

さらに、分割後にパーティションをマージすると、Data Grid はクラスターに全ノードが存在することを確認しようとします。ノードからの応答時間にタイムアウトまたは上限が適用されるため、クラスタービューをマージする操作は遅延します。そのため、ネットワークの問題が発生したり、GC の時間が長くなる可能性があります。

別のシナリオとして、GC が原因でパーティション処理のパフォーマンスに影響を与える可能性がある場合に、GC が JVM を一時停止するとノードが 1 つ以上、クラスターから退出します。この問題が発生し、GC の完了後にノードが再開すると、ノードのクラスタートポロジーが古くなったり、競合したりする可能性があります。

マージポリシーが設定されている場合には、Data Grid はノードのマージ前に競合の解決を試行します。ただし、マージポリシーは、ノードに整合性のないハッシュがある場合にのみ使用されます。一貫性のあるハッシュは、セグメントごとに共通の所収者が 1 つ以上ある場合にはいずれも互換性があり、セグメント 1 つ以上に共通の所有者がない場合には、いずれも互換性がありません。

ノードに、古くなっているが、互換性と一貫性のあるハッシュが含まれる場合に、Data Grid は古くなったクラスタートポロジーを無視して、競合を解決しようとはしません。たとえば、ガベージコレクション (GC) が原因でクラスター内の 1 つのノードが一時停止している場合には、クラスター内の他のノードは一貫したハッシュから削除され、新しい所有者ノードに置き換えられます。**numOwners > 1** の場合には、古くなった一貫性のあるハッシュと新しい一貫性のあるハッシュでは、すべてのキーに共通の所有者があるので、互換性が確保され、Data Grid が競合解決プロセスを省略できるようにします。

2.13. クラスターのバックアップおよび障害復旧

全体的な CPU とメモリー割り当てに関しては通常、クロスサイトレプリケーションを実行する Data Grid クラスターは非対称です。サイジングに、クロスサイトレプリケーションを考慮する場合に、主な考慮事項は、クラスター間の状態遷移の操作による影響です。

たとえば、NYC の Data Grid クラスターはオフラインになり、クライアントは LON の Data Grid クラスターに切り替えるとします。NYC のクラスターがオンラインに戻ると、LON から NYC への状態遷移が発生します。この操作はクライアントから古い内容を読み取らないようにすることができますが、状態遷移を受信するクラスターのパフォーマンスが低下します。

状態遷移操作が必要とする処理量が増加した分をクラスター全体に分散できます。ただし、状態遷移操作によるパフォーマンスへの影響は、データセットのタイプやサイズなどの環境や要素により全く異なります。

Active/Active デプロイメントの競合解決

Data Grid は、複数のサイトがクライアント要求を処理すると (Active/Active のサイト設定と呼ばれる)、同時書き込み操作による競合を検出します。

次の例では、**LON**データセンターと**NYC**データセンターで実行されている Data Grid クラスターで、同時書き込みの結果、エントリーが競合する様子を示しています。

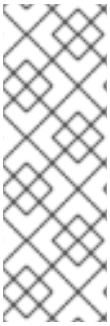
```
          LON       NYC

k1=(n/a)   0,0       0,0

k1=2       1,0  -->  1,0   k1=2

k1=3       1,1  <--  1,1   k1=3

k1=5       2,1       1,2   k1=8

               -->  2,1 (conflict)
(conflict)  1,2  <--
```

Active/Active のサイト設定では、同時書き込みがデッドロックになり、データが損失するので、同期バックアップストラテジーを使用しないでください。非同期バックアップ戦略 (**strategy=async**) を使用すると、Data Grid では、同時書き込みの処理に、クロスサイトマージポリシーを選択できます。

パフォーマンスの面では、Data Grid が競合解決に使用するマージポリシーでは、追加の計算が必要ですが、通常は大きなペナルティーは発生しません。たとえば、デフォルトのクロスサイトマージポリシーは、辞書式比較 (または文字列の比較) を使用しており、完了するまでに数ナノ秒しかかかりませ

ん。

Data Grid では、クロスサイトマージポリシーの **XSiteEntryMergePolicy** SPI も提供します。Data Grid がカスタム実装との競合を解決するように設定するには、常にパフォーマンスを監視して悪影響がないかを監視する必要があります。



注記

XSiteEntryMergePolicy SPI は、非ブロッキングスレッドプールのマージポリシーをすべて呼び出します。ブロッキングのカスタムマージポリシーを実装すると、スレッドプールをすべて使い切ってしまう可能性があります。

複雑なポリシーまたはブロックポリシーを異なるスレッドに委譲する必要があり、この実装では、マージポリシーが他のスレッドで完了した時点で、終了する **CompletionStage** を返す必要があります。

2.14. コードの実行およびデータ処理

分散キャッシングの利点の 1 つは、各ホストからのコンピュートリソースを使用して、大規模なスケーリングデータ処理をより効率的に実行できることです。Data Grid で直接処理ロジックを実行することで、ワークロードを複数の JVM インスタンスに分散します。コードは、データを格納しているのと同じメモリー領域でも実行されるので、エントリーの反復処理速度を早めることができます。

Data Grid のデプロイメントへのパフォーマンスの影響については、コードの実行状況により完全に異なります。より複雑な処理操作を行うと、パフォーマンスが低下するため、慎重に計画を立てて、Data Grid クラスターでコードを実行するようにする必要があります。コードをテストし、小規模なサンプルデータセットで複数実行を試すことから開始します。メトリックを収集後には、最適化の特定を開始し、実行中のコードのパフォーマンスに対する影響を理解できます。

確実に考慮するべき点として、処理を長時間にわたり実行すると、通常の読み取りおよび書き込みの操作に悪影響を及ぼし始めます。したがって、デプロイメントを時間とともに監視し、パフォーマンスを継続的に評価することが重要です。

埋め込みキャッシュ
Data Grid は埋め込みのキャッシュで、データと同じメモリー領域でコードを実行できる API を 2 つ提供します。

ClusterExecutor API

1 つ以上のキャッシュエントリーの反復処理など、Cache Manager ですべての操作を実行し、Data Grid ノードに基づいた処理を行うことができます。

CacheStream API

コレクションで操作を実行し、データを基に処理できるようにします。

1 つのノード、ノードのグループ、または特定の地理的リージョン内の全ノードで操作を実行する場合は、クラスターの実行を使用する必要があります。データセット全体に対して正しい結果を保証する操作を実行する場合には、分散ストリームを使用するオプションがより効果的です。

クラスターの実行

```
ClusterExecutor clusterExecutor = cacheManager.executor();
clusterExecutor.singleNodeSubmission().filterTargets(policy);
for (int i = 0; i < invocations; ++i) {
  clusterExecutor.submitConsumer((cacheManager) -> {
    TransportConfiguration tc =
```

```
      cacheManager.getCacheManagerConfiguration().transport();
      return tc.siteId() + tc.rackId() + tc.machineId();
   }, triConsumer).get(10, TimeUnit.SECONDS);
}
```

CacheStream

```
Map<Object, String> jbossValues =
cache.entrySet().stream()
    .filter(e -> e.getValue().contains("JBoss"))
    .collect(Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue));
```

関連情報

- org.infinispan.manager.ClusterExecutor
- org.infinispan.CacheStream

リモートキャッシュ

リモートキャッシュの場合、Data Grid は **ServerTask** API を提供し、カスタム Java 実装を Data Grid Server に登録し、Hot Rod で **execute()** メソッドを呼び出すか、Data Grid コマンドラインインターフェイス (CLI) を使用して、プログラムを使用してタスクを実行できます。1 つの Data Grid Server インスタンスだけで、またはクラスター内のすべてのサーバーインスタンスでタスクを実行できます。

2.15. クライアントトラフィック

リモート Data Grid クラスターのサイジング時には、エントリーの数およびサイズを計算する必要がありますが、クライアントトラフィックの量も計算します。Data Grid は、データを格納し、クライアントの読み取り要求と書き込み要求をタイムリーに処理するために十分な RAM が必要です。

レイテンシーに影響し、応答時間を決定する要因は多数あります。たとえば、キー/値のペアのサイズは、リモートキャッシュの応答時間に影響します。リモートキャッシュのパフォーマンスに影響を与える他の要因には、クラスターが受信する 1 秒あたりの要求数、クライアントの数、書き込み操作に対する読み取り動作の比率などが含まれます。

第3章 OPENSHIFT での DATA GRID のベンチマーク

OpenShift で実行している Data Grid クラスターの場合には、Red Hat は Hyperfoil を使用してパフォーマンスを測定することを推奨します。Hyperfoil は、分散サービスに対して正確なパフォーマンス結果を提供するベンチマークフレームワークです。

3.1. DATA GRID のベンチマーク

デプロイメントを設定して設定したら、Data Grid クラスターのベンチマークを開始して、パフォーマンスを分析し、測定します。ベンチマークにより、環境を調整し、Data Grid の設定を調整してパフォーマンスを最適化することができます。つまり、レイテンシーが低くなり、スループットが最も高いことを意味します。

パフォーマンスを最適化するのは、最終的な目標ではなく、継続的なプロセスであることに注意してください。ベンチマークテストで、Data Grid デプロイメントが希望するパフォーマンスレベルに達したことが示されている場合は、それらの結果が修正または常に有効になることを期待することはできません。

3.2. HYPERFOIL のインストール

Operator サブスクリプションを作成し、コマンドラインインターフェイス (CLI) が含まれる Hyperfoil ディストリビューションをダウンロードして、Red Hat OpenShift で Hyperfoil を設定します。

手順

1. OpenShift Web コンソールの **OperatorHub** を使用して Hyperfoil Operator サブスクリプションを作成します。

注記

Hyperfoil Operator は Community Operator として利用できます。

Red Hat は Hyperfoil Operator を認定せず、Data Grid と組み合わせてサポートを提供しません。Hyperfoil Operator をインストールすると、続行する前にコミュニティーバージョンに関する警告を確認するように求められます。

2. Hyperfoil リリースページ から最新の Hyperfoil バージョンをダウンロードします。

関連情報

- hyperfoil.io
- OpenShift への Hyperfoil のインストール

3.3. HYPERFOIL コントローラーの作成

Hyperfoil Controller を Red Hat OpenShift でインスタンス化することで、Hyperfoil コマンドラインインターフェイス (CLI) を使用してベンチマークテストをアップロードおよび実行できます。

前提条件

- Hyperfoil Operator サブスクリプションを作成します。

手順

1. **hyperfoil-controller.yaml** を定義します。

```
$ cat > hyperfoil-controller.yaml<<EOF
apiVersion: hyperfoil.io/v1alpha2
kind: Hyperfoil
metadata:
  name: hyperfoil
spec:
  version: latest
EOF
```

2. Hyperfoil Controller を適用します。

```
$ oc apply -f hyperfoil-controller.yaml
```

3. Hyperfoil CLI に接続するルートを取得します。

```
$ oc get routes

NAME        HOST/PORT
hyperfoil   hyperfoil-benchmark.apps.example.net
```

3.4. HYPERFOIL ベンチマークの実行

Hyperfoil を使用してベンチマークテストを実行し、Data Grid クラスターのパフォーマンスデータを収集します。

前提条件

- Hyperfoil Operator サブスクリプションを作成します。
- Red Hat OpenShift で Hyperfoil Controller をインスタンス化します。

手順

1. ベンチマークテストを作成します。

```
$ cat > hyperfoil-benchmark.yaml<<EOF
name: hotrod-benchmark
hotrod:
 # Replace <USERNAME>:<PASSWORD> with your Data Grid credentials.
 # Replace <SERVICE_HOSTNAME>:<PORT> with the host name and port for Data Grid.
 - uri: hotrod://<USERNAME>:<PASSWORD>@<SERVICE_HOSTNAME>:<PORT>
   caches:
    # Replace <CACHE-NAME> with the name of your Data Grid cache.
    - <CACHE-NAME>
agents:
  agent-1:
  agent-2:
  agent-3:
  agent-4:
  agent-5:
```

```
phases:
- rampupPut:
    increasingRate:
      duration: 10s
      initialUsersPerSec: 100
      targetUsersPerSec: 200
      maxSessions: 300
      scenario: &put
      - putData:
        - randomInt: cacheKey <- 1 .. 40000
        - randomUUID: cacheValue
        - hotrodRequest:
            # Replace <CACHE-NAME> with the name of your Data Grid cache.
            put: <CACHE-NAME>
            key: key-${cacheKey}
            value: value-${cacheValue}
- rampupGet:
    increasingRate:
      duration: 10s
      initialUsersPerSec: 100
      targetUsersPerSec: 200
      maxSessions: 300
      scenario: &get
      - getData:
        - randomInt: cacheKey <- 1 .. 40000
        - hotrodRequest:
            # Replace <CACHE-NAME> with the name of your Data Grid cache.
            get: <CACHE-NAME>
            key: key-${cacheKey}
- doPut:
    constantRate:
      startAfter: rampupPut
      duration: 5m
      usersPerSec: 10000
      maxSessions: 11000
      scenario: *put
- doGet:
    constantRate:
      startAfter: rampupGet
      duration: 5m
      usersPerSec: 40000
      maxSessions: 41000
      scenario: *get
EOF
```

2. 任意のブラウザーでルートを開き、Hyperfoil CLI にアクセスします。
3. ベンチマークテストをアップロードします。
   a. **upload** コマンドを実行します。

      ```
      [hyperfoil]$ upload
      ```

   b. **Select benchmark file** をクリックしてから、ファイルシステムのベンチマークテストに移動してアップロードします。

4. ベンチマークテストを実行します。

```
[hyperfoil]$ run hotrod-benchmark
```

5. ベンチマークテストの結果を取得します。

```
[hyperfoil]$ stats
```

3.5. HYPERFOIL ベンチマーク結果

Hyperfoil は、**stats** コマンドを使用してベンチマークの実行結果をテーブル形式で出力します。

```
[hyperfoil]$ stats
Total stats from run <run_id>
PHASE  METRIC  THROUGHPUT  REQUESTS  MEAN  p50  p90  p99  p99.9  p99.99  TIMEOUTS
ERRORS  BLOCKED
```

表3.1 列の説明

カラム	説明	値
PHASE	各実行について、Hyperfoil は 2 つのフェーズで Data Grid クラスターに **GET** 要求と **PUT** 要求を実行します。	**doGet** または **doPut**
METRIC	実行フェーズの両方で、Hyperfoil は **GET** および **PUT** 要求ごとにメトリックを収集します。	**getData** または **putData** のいずれか
THROUGHPUT	1 秒あたりのリクエストの総数を取得します。	Number
REQUESTS	実行の各フェーズ中に操作の合計数をキャプチャーします。	Number
MEAN	**GET** または **PUT** 操作の完了の平均時間を取得します。	時間 (**ms**)
p50	50 パーセントの要求が完了するのにかかる時間を記録します。	時間 (**ms**)
p90	90 パーセントの要求が完了するのにかかる時間を記録します。	時間 (**ms**)
p99	99 パーセント要求の完了にかかった時間を記録します。	時間 (**ms**)
p99.9	99.9 パーセントの要求が完了するのにかかる時間を記録します。	時間 (**ms**)

カラム	説明	値
p99.99	完了する要求の 99.99 パーセントにかかる時間を記録します。	時間 (**ms**)
TIMEOUTS	実行の各フェーズで操作に対して発生するタイムアウトの合計数を取得します。	Number
ERRORS	実行の各フェーズ中に発生したエラーの合計数を取得します。	Number
BLOCKED	ブロックまたは完了されなかった操作の総数を取得します。	Number