Red Hat Streams for Apache Kafka 2.7

Kafka 設定のチューニング

Kafka 設定プロパティーを使用してデータのストリーミングを最適化する

Last Updated: 2024-05-29

Red Hat Streams for Apache Kafka 2.7 Kafka 設定のチューニング

Kafka 設定プロパティーを使用してデータのストリーミングを最適化する

法律上の通知

概要

Kafka 設定プロパティーを使用して、Kafka ブローカー、プロデューサー、およびコンシューマーの操作を微調整します。

目次

はじめに

RED HAT ドキュメントへのフィードバック (英語のみ)

Red Hat ドキュメントに関するご意見やご感想をお寄せください。

改善を提案するには、Jira 課題を作成し、変更案についてご説明ください。ご要望に迅速に対応できるよう、できるだけ詳細にご記入ください。

前提条件

- Red Hat カスタマーポータルのアカウントがある。このアカウントを使用すると、Red Hat Jira Software インスタンスにログインできます。
  アカウントをお持ちでない場合は、アカウントを作成するように求められます。

手順

1. 以下の Create issue をクリックします。
2. **Summary** テキストボックスに、問題の簡単な説明を入力します。
3. **Description** テキストボックスに、次の情報を入力します。
   - 問題が見つかったページの URL
   - 問題の詳細情報
     他のフィールドの情報はデフォルト値のままにすることができます。
4. レポーター名を追加します。
5. **Create** をクリックして、Jira 課題をドキュメントチームに送信します。

フィードバックの提供にご協力いただきありがとうございました。

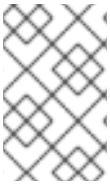
第1章 KAFKA チューニングの概要

Kafka デプロイメントのパフォーマンスを微調整するには、特定の要件に応じてさまざまな設定プロパティーを最適化する必要があります。このセクションでは、Kafka ブローカー、プロデューサー、およびコンシューマーで使用できる一般的な設定オプションについて紹介します。

Kafka を機能させるには最小限の設定セットが必要ですが、Kafka のプロパティーにより広範囲にわたる調整が可能になります。設定プロパティーを使用して、レイテンシー、スループット、および全体的な効率を向上させ、Kafka のデプロイメントがアプリケーションの要求を満たすようにすることができます。

効果的なチューニングを行うには、体系的なアプローチを採用してください。まず、関連するメトリクスを分析して、潜在的なボトルネックや改善の余地がある領域を特定します。設定パラメーターを繰り返し調整し、パフォーマンスメトリクスへの影響を監視して、それに応じて設定を調整します。

Apache Kafka 設定プロパティーの詳細は、Apache Kafka のドキュメント を参照してください。



注記

ここで提供されるガイダンスは、Kafka デプロイメントを調整するための出発点となります。最適な設定を見つけるには、ワークロード、インフラストラクチャー、パフォーマンス目標などの要因に依存します。

1.1. プロパティーと値のマッピング

設定プロパティーを指定する方法は、デプロイメントの種類によって異なります。OCP に Streams for Apache Kafka をデプロイした場合は、**Kafka** リソースを使用して、**config** プロパティーを介して Kafka ブローカーの設定を追加できます。Streams for Apache Kafka on RHEL では、設定を環境変数としてプロパティーファイルに追加します。

カスタムリソースに **config** プロパティーを追加するときは、コロン (':') を使用してプロパティーと値をマップします。

カスタムリソースの設定例

```
num.partitions:1
```

プロパティーを環境変数として追加する場合は、等号 (=) を使用してプロパティーおよび値をマップします。

環境変数としての設定例

```
num.partitions=1
```

注記

本ガイドのいくつかの例は、特に Streams for Apache Kafka on OpenShift のリソース設定を示している場合があります。ただし、RHEL で Streams for Apache Kafka を使用する場合、プロパティーは環境変数として同等に適用されます。

1.2. チューニングに役立つツール

次のツールは、Kafka のチューニングに役立ちます。

- Cruise Control。クラスターのリバランスを評価および実装するために使用できる最適化の提案を生成します。
- Kafka Static Quota プラグイン。ブローカーに制限を設定します。
- ラック設定。ブローカーパーティションをラック全体に広げ、コンシューマーが最も近いレプリカからデータを取得できるようにします。

関連情報

これらのツールの詳細は、次のガイドを参照してください。

- OpenShift での Streams for Apache Kafka の設定
- KRaft モードの RHEL での Streams for Apache Kafka の使用
- ZooKeeper で RHEL での Streams for Apache Kafka の使用

第2章 マネージドブローカーの設定

Streams for Apache Kafka を OpenShift にデプロイするときは、**Kafka** カスタムリソースの **config** プロパティーを使用してブローカー設定を指定します。ただし、特定のブローカー設定オプションは、Streams for Apache Kafka によって直接管理されます。

そのため、OpenShift で Streams for Apache Kafka を使用している場合は、以下のオプションを設定できません。

- Kafka ブローカーの ID を指定する **broker.id**
- ログデータ用の **log.dirs** ディレクトリー
- Kafka と ZooKeeper を接続する **zookeeper.connect** 設定
- Kafka クラスターをクライアントに公開するための **listeners**
- ユーザーが実行するアクションを許可または拒否する **authorization** メカニズム
- Kafka へのアクセスを必要とするユーザーのアイデンティティーを証明する **authentication** メカニズム

ブローカー ID は 0 (ゼロ) から開始し、ブローカーレプリカの数に対応します。ログディレクトリーは、**Kafka** カスタムリソースの **spec.kafka.storage** 設定に基づき、**/var/lib/kafka/data/kafka-logIDX** にマウントされます。**IDX** は Kafka ブローカー Pod インデックスです。

除外項目の一覧は、**KafkaClusterSpec** schema reference を参照してください。

RHEL で Streams for Apache Kafka を使用する場合、これらの除外は適用されません。この場合、ブローカーを識別し、セキュアなアクセスを提供するには、これらのプロパティーをブローカーの基本設定に追加する必要があります。

RHEL での Streams for Apache Kafka のブローカー設定の例

```
# ...
broker.id = 1
log.dirs = /var/lib/kafka
zookeeper.connect = zoo1.my-domain.com:2181,zoo2.my-domain.com:2181,zoo3.my-
domain.com:2181
listeners = internal-1://:9092
authorizer.class.name = kafka.security.auth.SimpleAclAuthorizer
ssl.truststore.location = /path/to/truststore.jks
ssl.truststore.password = 123456
ssl.client.auth = required
# ...
```

関連情報

- OCP での Kafka の設定
- KRaft モードの RHEL での Streams for Apache Kafka の使用
- ZooKeeper で RHEL での Streams for Apache Kafka の使用

第3章 KAFKA ブローカー設定のチューニング

設定プロパティーを使用して、Kafka ブローカーのパフォーマンスを最適化します。Streams for Apache Kafka によって直接管理されるプロパティーを除き、標準の Kafka ブローカー設定オプションを使用できます。

3.1. 基本的なブローカー設定

通常のブローカー設定には、トピック、スレッド、およびログに関連するプロパティーの設定が含まれます。

基本的なブローカープロパティー

```
# ...
num.partitions=1
default.replication.factor=3
offsets.topic.replication.factor=3
transaction.state.log.replication.factor=3
transaction.state.log.min.isr=2
log.retention.hours=168
log.segment.bytes=1073741824
log.retention.check.interval.ms=300000
num.network.threads=3
num.io.threads=8
num.recovery.threads.per.data.dir=1
socket.send.buffer.bytes=102400
socket.receive.buffer.bytes=102400
socket.request.max.bytes=104857600
group.initial.rebalance.delay.ms=0
zookeeper.connection.timeout.ms=6000
# ...
```

3.2. 高可用性のためのトピックの複製

基本的なトピックプロパティーは、トピックのデフォルト数のパーティションおよびレプリケーション係数を設定します。これは、トピックが自動的に作成される場合を含め、これらのプロパティーを明示的に設定せずに作成されたトピックに適用されます。

```
# ...
num.partitions=1
auto.create.topics.enable=false
default.replication.factor=3
min.insync.replicas=2
replica.fetch.max.bytes=1048576
# ...
```

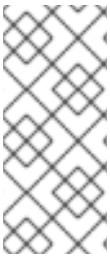
高可用性環境の場合は、トピックに対してレプリケーション係数を 3 以上に引き上げ、必要な同期レプリカの最小数をレプリケーション係数より 1 少なく設定することを推奨します。

auto.create.topics.enable プロパティーはデフォルトで有効になっており、存在しないトピックがプロデューサーおよびコンシューマーによって必要になると自動的に作成されます。トピックの自動作成を使用する場合は、**num.partitions** を使用してトピックのデフォルトのパーティション数を設定できます。しかし、一般的には、このプロパティーは無効にして、明示的にトピックを作成することでトピックを詳細に制御できるようにします。

また、データの持続性 を確保するために、topic の設定で **min.insync.replicas** を設定し、producer の設定で **acks=all** を使用してメッセージ配信の確認を行う必要があります。

replica.fetch.max.bytes を使用して、リーダーパーティションを複製する各フォロワーが取得したメッセージの最大サイズ (バイト単位) を設定します。この値は、平均のメッセージサイズおよびスループットに応じて変更します。読み取り/書き込みバッファーに必要なメモリー割り当ての合計を考慮すると、利用可能なメモリーも、すべてのフォロワーで乗算した時のレプリケートメッセージの最大サイズに対応できる必要があります。

delete.topic.enable プロパティーはデフォルトで有効になっており、トピックの削除を許可します。実稼働環境では、誤ってトピックが削除され、データが失われるのを防ぐために、このプロパティーを無効にする必要があります。ただし、トピックを一時的に有効にして、トピックを削除してから再度無効にできます。



注記

OpenShift で Streams for Apache Kafka を実行する場合、Topic Operator は operator スタイルのトピック管理を提供できます。**KafkaTopic** リソースを使用してトピックを作成できます。**KafkaTopic** リソースを使用して作成されたトピックの場合、レプリケーション係数は **spec.replicas** で設定されます。**delete.topic.enable** が有効になっている場合は、**KafkaTopic** リソースを使用してトピックを削除できます。

```
# ...
auto.create.topics.enable=false
delete.topic.enable=true
# ...
```

3.3. トランザクションおよびコミットの内部トピック設定

トランザクションを使用 してプロデューサーからのパーティションへのアトミック書き込みを有効にする場合、トランザクションの状態は内部 **__transaction_state** トピックに保存されます。デフォルトでは、ブローカーはレプリケーション係数が 3 で設定され、このトピックでは少なくとも 2 つの同期レプリカが設定されます。つまり、Kafka クラスターには少なくとも 3 つのブローカーが必要になります。

```
# ...
transaction.state.log.replication.factor=3
transaction.state.log.min.isr=2
# ...
```

同様に、コンシューマーの状態を格納する内部 **__consumer_offsets** トピックには、パーティションおよびレプリケーション係数のデフォルト設定があります。

```
# ...
offsets.topic.num.partitions=50
offsets.topic.replication.factor=3
# ...
```

実稼働ではこれらの設定を下げないでください。実稼働 環境で設定を大きくすることができます。例外として、単一ブローカーの **テスト環境** の設定を下げる必要がある場合があります。

3.4. I/O スレッドの増加によるリクエスト処理スループットの向上

ネットワークスレッドは、クライアントアプリケーションからのリクエストの生成や取得など、Kafka クラスターへのリクエストを処理します。生成リクエストはリクエストキューに配置されます。レスポンスはレスポンスキューに配置されます。

リスナーごとのネットワークスレッドの数は、レプリケーション係数と、Kafka クラスターと、対話するクライアントプロデューサーおよびコンシューマーからのアクティビティーのレベルを反映する必要があります。リクエストが多い場合は、スレッドがアイドル状態である時間を使用してスレッドの数を増やし、スレッドを追加するタイミングを決定できます。

輻輳を軽減し、リクエストトラフィックを規制するには、リクエストキューで許可されるリクエスト数を制限できます。リクエストキューがいっぱいになると、すべての着信トラフィックがブロックされます。

I/O スレッドはリクエストキューからリクエストを選択して処理します。スレッド数を増やすとスループットが向上しますが、CPU のコアの数とおよびディスク帯域幅により、実用的な上限が決まります。最低でも、I/O スレッドの数はストレージボリュームの数と同じでなければなりません。

```
# ...
num.network.threads=3 1
queued.max.requests=500 2
num.io.threads=8 3
num.recovery.threads.per.data.dir=4 4
# ...
```

1. Kafka クラスターのネットワークスレッドの数。
2. リクエストキューで許可されるリクエストの数。
3. Kafka ブローカーの I/O スレッドの数。
4. 起動時のログの読み込みおよびシャットダウン時のフラッシュに使用されるスレッドの数。コア数以上の値に設定してみてください。

すべてのブローカーのスレッドプールへの設定の更新は、クラスターレベルで動的に発生する可能性があります。これらの更新は、現在のサイズの半分から現在のサイズの 2 倍までに制限されます。

ヒント

次の Kafka ブローカーメトリクスは、必要なスレッド数を計算するのに役立ちます。

- **kafka.network:type=SocketServer,name=NetworkProcessorAvgIdlePercent** は、ネットワークスレッドがアイドル状態である平均時間に関する指標をパーセンテージで提供します。
- **kafka.server:type=KafkaRequestHandlerPool,name=RequestHandlerAvgIdlePercent** は、I/O スレッドがアイドル状態である平均時間に関する指標をパーセンテージで提供します。

アイドル時間が 0% の場合は、すべてのリソースが使用中であるため、スレッドを追加すると効果がある可能性があります。アイドル時間が 30% を下回ると、パフォーマンスが低下し始める可能性があります。

ディスクの数によりスレッドが遅くなるか、制限が課される場合には、ネットワークリクエストのバッファーのサイズを増やしてスループットを向上させることができます。

```
# ...
replica.socket.receive.buffer.bytes=65536
# ...
```

また、Kafka が受信可能な最大バイト数も増やします。

```
# ...
socket.request.max.bytes=104857600
# ...
```

3.5. レイテンシーの高い接続に対する帯域幅の引き上げ

Kafka はデータをバッチ処理して、データセンター間の接続など、Kafka からクライアントへのレイテンシーの高い接続で妥当なスループットを実現します。ただし、レイテンシーの高さが問題である場合、メッセージを送受信するためのバッファーのサイズを増やすことができます。

```
# ...
socket.send.buffer.bytes=1048576
socket.receive.buffer.bytes=1048576
# ...
```

帯域幅遅延積 の計算を使用して、バッファーの最適なサイズを見積もることができます。これは、リンクの最大帯域幅 (バイト/秒) にラウンドトリップ遅延 (秒) を乗算して、最大スループットの維持に必要なバッファーの大きさを見積もります。

3.6. 削除および圧縮ポリシーを使用した KAFKA ログの管理

Kafka はログに依存してメッセージデータを保存します。ログは一連のセグメントで設定され、各セグメントはオフセットベースおよびタイムスタンプベースのインデックスに関連付けられます。新しいメッセージは **アクティブ** なセグメントに書き込まれ、その後変更されません。コンシューマーからのフェッチ要求を処理するときに、セグメントが読み取られます。定期的に、アクティブセグメントが **ロール** されて読み取り専用になり、それを置き換えるために新しいアクティブセグメントが作成されます。トピックパーティションごとに 1 つのアクティブなセグメントのみがあります。古いセグメントは、削除の対象となるまで保持されます。

ブローカーレベルの設定により、ログセグメントの最大サイズ (バイト単位) とアクティブなセグメントがロールされるまでの時間 (ミリ秒単位) が決まります。

```
# ...
log.segment.bytes=1073741824
log.roll.ms=604800000
# ...
```

これらの設定は、**segment.bytes** および **segment.ms** を使用してトピックレベルでオーバーライドできます。これらの値を下げるか上げるかの選択は、セグメント削除のポリシーによって異なります。サイズが大きいほど、アクティブセグメントに含まれるメッセージが多くなり、ロールされる頻度が少なくなります。セグメントが削除の対象となる頻度も低くなります。

Kafka では、ログクリーンアップポリシーによってログデータの管理方法が決まります。ほとんどの場合、クラスターレベルでデフォルト設定を変更する必要はないため、**delete** クリーンアップポリシーが指定され、**compact** クリーンアップポリシーで使用されるログクリーナーが有効になります。

```
# ...
```

```
log.cleanup.policy=delete
log.cleaner.enable=true
# ...
```

Delete クリーンアップポリシー

Delete クリーンアップポリシーは、全トピックを対象にするデフォルトのクラスター全体のポリシーです。このポリシーは、特定のトピックレベルのポリシーが設定されていないトピックに適用されます。Kafka は、時間ベースまたはサイズベースのログ保持制限に基づいて古いセグメントを削除します。

Compact クリーンアップポリシー

Compact クリーンアップポリシーは通常、トピックレベルのポリシー (**cleanup.policy=compact**) として設定されます。Kafka のログクリーナーは、特定のトピックに圧縮を適用し、トピック内のキーの最新の値のみを保持します。両方のポリシー (**cleanup.policy=compact,delete**) を使用するようにトピックを設定することもできます。

削除ポリシーの保持制限の設定

Delete クリーンアップポリシーは、データを保持したログの管理に対応します。このポリシーは、データを永久に保持する必要がない場合に適しています。時間ベースまたはサイズベースのログ保持ポリシーとクリーンアップポリシーを確立して、ログを制限した状態に保つことができます。

ログ保持ポリシーが使用される場合、保持制限に達すると、アクティブではないログセグメントが削除されます。古いセグメントを削除すると、ディスク容量の超過を防ぐことができます。

期間ベースのログ保持の場合は、時間、分、またミリ秒ベースで保持期間を設定します。

```
# ...
log.retention.ms=1680000
# ...
```

保持期間は、メッセージがセグメントに追加された時間に基づいています。Kafka は、セグメント内の最新のメッセージのタイムスタンプを使用して、そのセグメントの有効期限が切れているかどうかを判断します。ミリ秒設定は分設定よりも優先され、分設定は時間設定おりも優先されます。分とミリ秒の設定はデフォルトで null ですが、3 つのオプションにより、保持するデータを実質的に制御できます。動的に更新できるのは 3 つのプロパティーの 1 つだけであるため、ミリ秒設定を優先する必要があります。

log.retention.ms が -1 に設定されている場合には、ログ保持には時間制限が適用されないため、すべてのログが保持されます。ただし、この設定は、ディスクがいっぱいになるなど、修正の困難な問題が発生する可能性があるため、通常は推奨されません。

サイズベースのログ保持の場合、最小ログサイズ (バイト単位) を指定します。

```
# ...
log.retention.bytes=1073741824
# ...
```

これは、Kafka により、少なくとも指定された量の利用可能なログデータを常に存在させることができます。

たとえば、**log.retention.bytes を** 1000 に設定し、**log.segment.bytes を** 300 に設定した場合、Kafka は 4 セグメントとアクティブセグメントを保持し、少なくとも 1000 バイトを使用できるようにします。アクティブなセグメントがいっぱいになり、新しいセグメントが作成されると、最も古いセグメントが削除されます。この時点で、ディスク上のサイズは指定された 1000 バイトを超える可能性があり、1200 ~ 1500 バイトの範囲になる可能性があります (インデックスファイルを除く)。

ログサイズの使用に関する潜在的な問題は、メッセージがセグメントに追加された時刻が考慮されていないことです。クリーンアップポリシーに時間ベースおよびサイズベースのログ保持を使用して、必要なバランスをとることができます。どちらのしきい値に最初に到達しても、クリーンアップがトリガーされます。

セグメントファイルがシステムから削除されるまでの遅延時間を追加するには、すべてのトピックに対してブローカーレベルで **log.segment.delete.lay.ms** を使用できます。

```
# ...
log.segment.delete.delay.ms=60000
# ...
```

または、トピックレベルで **file.delete.lay.ms** を設定します。

ログのクリーンアップをチェックする頻度をミリ秒単位で設定します。

```
# ...
log.retention.check.interval.ms=300000
# ...
```

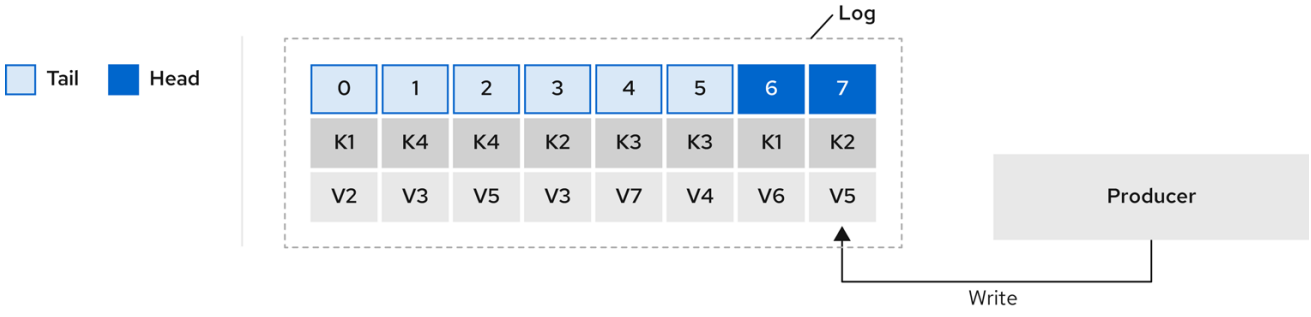
ログ保持設定に関連して、ログ保持チェックの間隔を調整します。保持サイズが小さいほど、より頻繁なチェックが必要になる場合があります。クリーンアップの頻度は、ディスクスペースを管理するのに十分な頻度である必要がありますが、ブローカーのパフォーマンスに影響を与えるほど頻度を上げてはなりません。

コンパクトポリシーを使用して最新のメッセージの保持

cleanup.policy=compact を設定してトピックのログ圧縮を有効にすると、Kafka はログクリーナーをバックグラウンドスレッドとして使用して圧縮を実行します。圧縮ポリシーは、各メッセージキーの最新のメッセージが保持されるようにし、古いバージョンのレコードを効果的に消去します。このポリシーは、メッセージ値が変更可能であり、最新の更新を保持する場合に適しています。

ログコンパクションにクリーンアップポリシーが設定されている場合、ログの **先頭** は標準の Kafka ログとして機能し、新しいメッセージへの書き込みが順番に追加されます。ログクリーナーが動作する圧縮ログの **末尾** で、ログの後半でキーが同じ別のレコードが発生した場合、レコードは削除されます。null 値を持つメッセージも削除されます。Kafka では、各キーの最新メッセージが保持されるようにしますが、圧縮されたログ全体に重複が含まれないようにすることができないので、圧縮を使用するには、関連するメッセージを識別するためのキーが必要です。

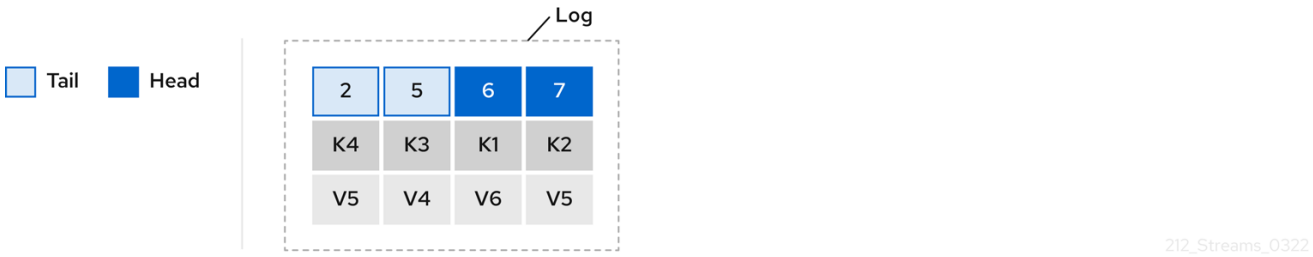
図3.1 コンパクション前のオフセットの位置によるキー値の書き込みを示すログ



キーを使用してメッセージを識別する Kafka 圧縮では、特定のメッセージキーのログ末尾に存在する最新のメッセージ (オフセットが最も大きい) が保持され、最終的にはキーが同じ、以前のメッセージが破棄されます。最新状態のメッセージは常に利用可能であり、その特定の以前のメッセージレコードは、

ログクリーナーの実行時に最終的に削除されます。メッセージを以前の状態に復元できます。周囲のレコードが削除されても、レコードは元のオフセットを保持します。その結果、末尾は連続しないオフセットを持つ可能性があります。末尾で使用できなくなったオフセットを消費すると、次に高いオフセットを持つレコードが見つかります。

図3.2 コンパクション後のログ



必要に応じて、圧縮プロセスに遅延を追加できます。

```
# ...
log.cleaner.delete.retention.ms=86400000
# ...
```

削除されたデータの保持期間は、データが完全に削除される前に、データが削除されたことに気付く時間を確保します。

特定のキーに関連するすべてのメッセージを削除するために、プロデューサーは廃棄 (tombstone) メッセージを送信できます。tombstone には null 値があり、そのキーの対応するメッセージが削除されたことをコンシューマーに知らせるマーカーとして機能します。しばらくすると、tombstone マーカーのみが保持されます。新しいメッセージが引き続き受信されると仮定すると、コンシューマーが削除を認識するのに十分な時間を確保できるように、マーカーは **log.cleaner.delete.retention.ms** で指定された期間保持されます。

クリーニングするログがない場合にクリーナーをスタンバイにする時間をミリ秒単位で設定することもできます。

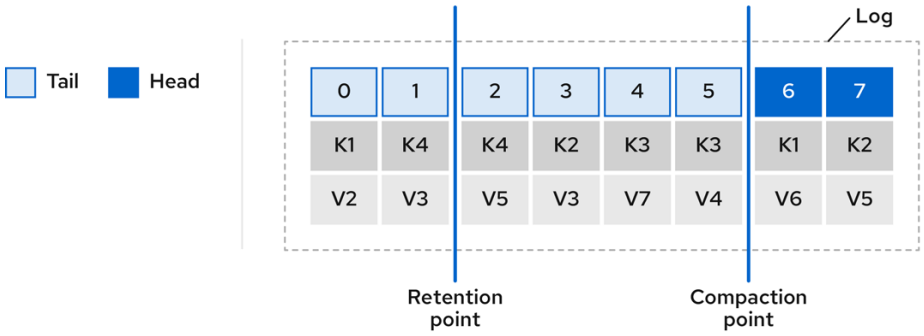
```
# ...
log.cleaner.backoff.ms=15000
# ...
```

圧縮ポリシーと削除ポリシーを組み合わせて使用する

圧縮ポリシーのみを選択すると、ログが任意に大きくなる可能性があります。このような場合、ログを圧縮して削除するようにトピックのクリーンアップポリシーを設定できます。Kafka はログ圧縮を適用し、古いバージョンのレコードを削除し、各キーの最新バージョンのみを保持します。Kafka は、指定された時間ベースまたはサイズベースのログ保持設定に基づいてレコードも削除します。

たとえば、次の図では、特定のメッセージキーの最新のメッセージ (オフセットが最も大きい) のみが圧縮ポイントまで保持されます。保持ポイントまでのレコードが残っている場合は、削除されます。この場合、圧縮プロセスではすべての重複が削除されます。

図3.3 ログ保持ポイントおよびコンパクションポイント



3.7. 圧縮の効率的なディスク使用率の管理

圧縮ポリシーとログクリーナーを使用して Kafka でトピックログを処理する場合は、メモリー割り当ての最適化を検討してください。

deduplication プロパティー (**dedupe.buffer.size**) を使用してメモリー割り当てを微調整でき、すべてのログクリーナースレッド全体でクリーンアップタスクに割り当てられる合計メモリーが決まります。さらに、**buffer.load.factor** プロパティーでパーセンテージを定義することで、最大メモリー使用量制限を設定できます。

```
# ...
log.cleaner.dedupe.buffer.size=134217728
log.cleaner.io.buffer.load.factor=0.9
# ...
```

各ログエントリーは正確に 24 バイトを使用するため、バッファーが 1 回の実行で処理できるログエントリーの数を計算し、それに応じて設定を調整できます。

可能であれば、ログのクリーニング時間を短縮する場合は、ログクリーナースレッドの数を増やすことを検討してください。

```
# ...
log.cleaner.threads=8
# ...
```

ディスク帯域幅の使用率が 100% で問題が発生している場合は、読み書き操作の合計が、操作を実行するディスクの機能に基づいて指定された値の 2 倍未満になるように、ログクリーナーの I/O を調整できます。

```
# ...
log.cleaner.io.max.bytes.per.second=1.7976931348623157E308
# ...
```

3.8. メッセージデータのログフラッシュの制御

一般に、明示的なフラッシュしきい値を設定せず、オペレーティングシステムにデフォルト設定を使用してバックグラウンドフラッシュを実行させることを推奨します。パーティションレプリケーションは、障害が発生したブローカーが同期レプリカから回復できるため、単一のディスクへの書き込みよりもデータの持続性が優れています。

ログフラッシュプロパティーは、キャッシュされたメッセージデータのディスクへの定期的な書き込みを制御します。スケジューラーは、ログキャッシュのチェック頻度をミリ秒単位で指定します。

```
# ...
log.flush.scheduler.interval.ms=2000
# ...
```

メッセージがメモリーに保持される最大時間と、ディスクに書き込む前にログに記録されるメッセージの最大数に基づいて、フラッシュの頻度を制御できます。

```
# ...
log.flush.interval.ms=50000
log.flush.interval.messages=100000
# ...
```

フラッシュ間の待機時間には、チェックを行う時間と、フラッシュが実行される前の指定された間隔が含まれます。フラッシュの頻度を増やすと、スループットに影響を及ぼす可能性があります。

アプリケーションフラッシュ管理を使用しており、より高速なディスクを使用している場合には、フラッシュしきい値を低く設定することが適切な場合があります。

3.9. 可用性のためのパーティションリバランス

フォールトトレランスのために、パーティションはブローカー間全体で複製できます。指定したパーティションでは、1つのブローカーがリーダーに選出され、すべての生成リクエストを処理します (ログへの書き込み)。他のブローカーのパーティションフォロワーは、リーダーに障害が発生した場合のデータの信頼性のために、パーティションリーダーのパーティションデータを複製します。

通常、フォロワーはクライアントを提供しませんが、**rack** 設定は、Kafka クラスターが複数のデータセンターにまたがる場合に最も近いレプリカからメッセージを消費できます。フォロワーは、パーティションリーダーからのメッセージを複製して、リーダーに障害が発生した場合に回復できるようにするためにのみ動作します。リカバリーには、同期のフォロワーが必要です。フォロワーは、フェッチリクエストをリーダーに送信することで同期を維持します。リーダーは、メッセージを順番にフォロワーに返します。フォロワーは、リーダーで最後にコミットされたメッセージに追いついた場合に、同期していると見なされます。リーダーは、フォロワーによってリクエストされた最後のオフセットを確認してこれをチェックします。クリーンでないリーダーエレクション (unclean leader election) が許可されない限り、非同期のフォロワーは通常、現在のリーダーが失敗した場合にリーダーとしての資格がありません。

フォロワーが同期していないと見なされるまでのラグタイムを調整できます。

```
# ...
replica.lag.time.max.ms=30000
# ...
```

ラグタイムは、メッセージをすべての同期レプリカにレプリケートする時間と、プロデューサーが確認レスポンスを待機する必要がある時間に上限を設定します。フォロワーがフェッチリクエストの作成に失敗し、指定されたラグタイム内に最新のメッセージに追いつくと、同期レプリカから削除されますラグタイムを短縮して、失敗したレプリカをより早く検出できますが、そうすると、不必要に同期から外れるフォロワーの数が増える可能性があります。適切なラグタイムの値は、ネットワークレイテンシーとブローカーのディスク帯域幅の両方に依存します。

リーダーパーティションが利用できなくなると、同期レプリカの1つが新しいリーダーとして選択されます。パーティションにあるレプリカのリストの最初のブローカーは、**優先** リーダーと呼ばれます。デフォルトでは、Kafka はリーダー分散の定期的なチェックに基づいて自動パーティションリーダーリバ

ランスに対して有効になっています。つまり、Kafka は優先リーダーが **現在** のリーダーであるかどうかを確認します。リバランスにより、リーダーがブローカー間で均等に分散され、ブローカーがオーバーロードされないようにします。

Cruise Control for Streams for Apache Kafka を使用すると、クラスター全体で負荷を均等に分散するブローカーへのレプリカの割り当てを把握できます。その計算では、リーダーとフォロワーで発生するさまざまな負荷が考慮されています。リーダーが失敗すると、残りのブローカーが追加のパーティションをリードするという余分な作業が発生するため、Kafka クラスターのバランスに影響を与えます。

Cruise Control で検出された割り当てが実際にバランスが取れている場合には、優先リーダーがパーティションのリーダーとなる必要があります。Kafka は、優先リーダーが使用されていることを自動的に確認し (可能な場合)、必要に応じて現在のリーダーを変更します。これにより、クラスターは CruiseControl が検出した時のバランスの取れた状態に保たれます。

リバランスチェックの頻度 (秒単位) と、リバランスがトリガーされる前にブローカーで対応できる不均衡の最大率を制御できます。

```
#...
auto.leader.rebalance.enable=true
leader.imbalance.check.interval.seconds=300
leader.imbalance.per.broker.percentage=10
#...
```

ブローカーにおけるリーダーの不均衡の割合は、ブローカーが現在のリーダーであるパーティションの現在の数と、そのブローカーが優先リーダーであるパーティションの数との比率です。優先リーダーが同期状態にあることを前提として、割合をゼロにして、優先リーダーが常に選択されるようにすることができます。

リバランスのチェックでさらに制御が必要な場合は、自動リバランスを無効にすることができます。次に、**kafka-leader-election.sh** コマンドラインツールを使用してリバランスをトリガーするタイミングを選択できます。

注記

Streams for Apache Kafka で提供される Grafana ダッシュボードでは、レプリケーションが不十分なパーティションや、アクティブなリーダーを持たないパーティションのメトリクスが表示されます。

3.10. クリーンでないリーダーエレクション (UNCLEAN LEADER ELECTION)

同期レプリカへのリーダーエレクションは、データの損失がないことを保証するため、クリーンであると見なされます。これは、デフォルトで行われます。しかし、リーダーに選出する同期レプリカがない場合はどうなるのでしょうか。おそらく、ISR (同期レプリカ) には、リーダーのディスクが停止したときにのみリーダーが含まれていました。同期レプリカの最小数が設定されておらず、ハードドライブに取り返しのつかない障害が発生したときにパーティションリーダーと同期しているフォロワーがない場合、データはすでに失われています。それだけでなく、同期しているフォロワーがいないため、**新しいリーダーを選出することはできません。**

Kafka がリーダーの失敗を処理する方法を設定できます。

```
# ...
unclean.leader.election.enable=false
# ...
```

クリーンでないリーダーエレクションはデフォルトでは無効になっており、同期されていないレプリカはリーダーになれません。クリーンリーダーエレクションでは、古いリーダーが失われたときに ISR に他のブローカーがない場合に Kafka はそのリーダーがオンラインに戻るまで待機してから、メッセージの読み書きが行われます。クリーンでないリーダーエレクションは、同期していないレプリカがリーダーになる可能性があることを意味しますが、メッセージが失われるリスクがあります。どちらを選択するかは、要件が可用性と持続性のどちらを優先するかによって異なります。

トピックレベルで特定のトピックのデフォルト設定を上書きできます。データ損失のリスクを許容できない場合は、デフォルト設定のままにします。

3.11. 不要なコンシューマーグループリバランスの回避

新しいコンシューマーグループに参加するコンシューマーの場合、ブローカーへの不要なリバランスを回避するために遅延を追加できます。

```
# ...
group.initial.rebalance.delay.ms=3000
# ...
```

この遅延は、コーディネーターがメンバーの参加を待つ期間です。遅延が長いほど、すべてのメンバーが時間内に参加し、リバランスを回避できる可能性が高くなります。ただし、遅延が発生すると、その期間が終了するまでグループは消費もできません。

第4章 KAFKA コンシューマー設定の調整

設定プロパティーを使用して、Kafka コンシューマーのパフォーマンスを最適化します。コンシューマーを調整する場合、最も重要なことは、取得するデータ量に効率的に対処できるようにすることです。プロデューサーのチューニングと同様に、コンシューマーが想定どおりに動作するまで、段階的に変更を加える必要があります。

コンシューマーをチューニングするときは、パフォーマンスと動作に大きな影響を与えるため、次の点を慎重に考慮してください。

スケーリング

コンシューマーグループは、複数のコンシューマーに負荷を分散することでメッセージの並列処理を可能にし、スケーラビリティーとスループットを向上させます。1 つのパーティションはコンシューマーグループ内の 1 つのコンシューマーにしか割り当てられないため、トピックパーティションの数により達成できる並列処理の最大レベルが決まります。

メッセージの順序付け

トピック内の絶対的な順序付けが重要な場合は、単一パーティションのトピックを使用してください。コンシューマーは、ブローカーにコミットされたのと同じ順序で単一パーティションのメッセージを監視します。つまり、Kafka は単一パーティションのメッセージの順序付けのみを保証します。ユーザーなどの個々のエンティティーに固有のイベントに関してメッセージの順序付けを維持することもできます。新しいエンティティーが作成された場合は、そのエンティティー専用の新しいトピックを作成できます。ユーザー ID などの一意の ID をメッセージキーとして使用し、同じキーを持つすべてのメッセージをトピック内の単一パーティションにルーティングできます。

オフセットリセットポリシー

適切なオフセットポリシーを設定すると、コンシューマーは目的の開始点からメッセージを消費し、適宜メッセージを処理できるようになります。Kafka のデフォルトのリセット値は **latest** であり、これはパーティションの最後から始まります。そのため、コンシューマーの動作とパーティションの状態によっては、一部のメッセージが失われる可能性があります。**auto.offset.reset** を **earliest** に設定すると、新しい **group.id** で接続するときに、すべてのメッセージがログの先頭から取得されるようになります。

アクセスのセキュリティー保護

Kafka へのセキュアなアクセスを管理 するユーザーアカウントを設定して、認証、暗号化、および承認のセキュリティー対策を実装します。

4.1. 基本的なコンシューマー設定

接続およびデシリアライザープロパティーはすべてのコンシューマーに必要です。通常、追跡用にクライアント ID を追加することが推奨されます。

コンシューマー設定では、後続の設定に関係なく、以下を行います。

- メッセージをスキップまたは再読み取りするようオフセットを変更しない限り、コンシューマーはメッセージを指定のオフセットから取得し、順番に消費します。
- オフセットはクラスターの別のブローカーに送信される可能性があるため、オフセットを Kafka にコミットした場合でも、ブローカーはコンシューマーがレスポンスを処理したかどうかを認識しません。

基本的なコンシューマー設定プロパティー

```
# ...
bootstrap.servers=localhost:9092 1
key.deserializer=org.apache.kafka.common.serialization.StringDeserializer 2
```

```
value.deserializer=org.apache.kafka.common.serialization.StringDeserializer 3
client.id=my-client 4
group.id=my-group-id 5
# ...
```

1. (必須) Kafka ブローカーの **host:port** ブートストラップサーバーアドレスを使用して、コンシューマーが Kafka クラスターに接続するよう指示しますコンシューマーはアドレスを使用して、クラスター内のすべてのブローカーを検出し、接続します。サーバーがダウンした場合に備えて、コンマ区切りリストを使用して 2 つまたは 3 つのアドレスを指定しますが、クラスター内のすべてのブローカーのリストを提供する必要はありません。ロードバランサーサービスを使用して Kafka クラスターを公開する場合、可用性はロードバランサーによって処理されるため、サービスのアドレスのみが必要になります。
2. (必須) Kafka ブローカーから取得されたバイトをメッセージキーに変換するデシリアライザー。
3. (必須) Kafka ブローカーから取得されたバイトをメッセージ値に変換するデシリアライザー。
4. (オプション) クライアントの論理名。リクエストのソースを特定するためにログおよびメトリックで使用されます。ID は、時間クォータの処理に基づいてコンシューマーにスロットリングを適用するために使用することもできます。
5. (条件) コンシューマーがコンシューマーグループに参加するには、グループ ID が **必要** です。

4.2. コンシューマーグループを使用したデータ消費のスケーリング

コンシューマーグループは、特定のトピックから 1 つまたは複数のプロデューサーによって生成される、典型的な大量のデータストリームを共有します。コンシューマーは **group.id** プロパティーを使用してグループ化されるため、メッセージをメンバー全体に分散できます。グループ内のコンシューマーの 1 つがリーダーを選択し、パーティションをグループのコンシューマーにどのように割り当てるかを決定します。各パーティションは 1 つのコンシューマーにのみ割り当てることができます。

コンシューマーの数がパーティションよりも少ない場合には、**group.id** が同じコンシューマーインスタンスを追加して、データの消費をスケーリングできます。コンシューマーをグループに追加して、パーティションの数より多くしても、スループットは改善されませんが、コンシューマーが機能しなくなったときに予備のコンシューマーを使用できます。より少ないコンシューマーでスループットの目標を達成できれば、リソースを節約できます。

同じコンシューマーグループのコンシューマーは、オフセットコミットとハートビートを同じブローカーに送信します。コンシューマーは、コンシューマーグループ内でのアクティビティーを示すために、Kafka ブローカーにハートビートを送信します。グループのコンシューマーの数が多いほど、ブローカーのリクエスト負荷が高くなります。

```
# ...
group.id=my-group-id 1
# ...
```

1. グループ ID を使用してコンシューマーグループにコンシューマーを追加します。

4.3. 適切なパーティション割り当てストラテジーの選択

適切なパーティション割り当てストラテジーを選択します。これにより、Kafka トピックパーティションがグループ内のコンシューマーインスタンス間でどのように分散されるかが決まります。

パーティションストラテジーは次のクラスでサポートされます。

- **org.apache.kafka.clients.consumer.RangeAssignor**
- **org.apache.kafka.clients.consumer.RoundRobinAssignor**
- **org.apache.kafka.clients.consumer.StickyAssignor**
- **org.apache.kafka.clients.consumer.CooperativeStickyAssignor**

partition.assignment.strategy コンシューマー設定プロパティーを使用してクラスを指定します。**range** 割り当てストラテジーは、パーティションの範囲を各コンシューマーに割り当てます。これは、関連するデータをまとめて処理する場合に便利です。

あるいは、コンシューマー間で均等にパーティションを分散する **round robin** 割り当てストラテジーを選択します。これは、並列処理を必要とする高スループットのシナリオに最適です。

より安定したパーティション割り当てを行うには、**sticky** ストラテジーや **cooperative sticky** ストラテジーを検討してください。Sticky ストラテジーは、可能な場合、リバランス中に割り当てられたパーティションを維持することを目的としています。以前コンシューマーに特定のパーティションが割り当てられていた場合、sticky ストラテジーは、リバランス後に同じコンシューマーでそれらの同じパーティションを保持することを優先し、別のコンシューマーに移動したパーティションのみを取り消して再割り当てします。パーティションの割り当てをそのままにしておくと、パーティション移動のオーバーヘッドが軽減されます。cooperative sticky ストラテジーは cooperative rebalance もサポートしており、再割り当てされていないパーティションから中断なく消費することを可能にします。

利用可能なストラテジーがデータに適合しない場合は、特定の要件に合わせたカスタムストラテジーを作成できます。

4.4. メッセージの順序の保証

Kafka ブローカーは、トピック、パーティション、およびオフセット位置のリストからメッセージを送信するようブローカーにリクエストするコンシューマーからフェッチリクエストを受け取ります。

コンシューマーは、ブローカーにコミットされたのと同じ順序でメッセージを単一のパーティションで監視します。つまり、Kafka は単一パーティションのメッセージ ***のみ*** 順序付けを保証します。逆に、コンシューマーが複数のパーティションからメッセージを消費している場合、コンシューマーによって監視される異なるパーティションのメッセージの順序は、必ずしも送信順序を反映しません。

1 つのトピックからメッセージを厳格に順序付ける場合は、コンシューマーごとに 1 つのパーティションを使用します。

4.5. スループットおよびレイテンシーに対するコンシューマーの最適化

クライアントアプリケーションが **KafkaConsumer.poll()** を呼び出すときに返されるメッセージの数を制御します。

fetch.max.wait.ms および **fetch.min.bytes** プロパティーを使用して、Kafka ブローカーからコンシューマーによって取得される最小データ量を増やします。時間ベースのバッチ処理は **fetch.max.wait.ms** を使用して設定され、サイズベースのバッチ処理は **fetch.min.bytes** を使用して設定されます。

コンシューマーまたはブローカーの CPU 使用率が高い場合、コンシューマーからのリクエストが多すぎる可能性があります。**fetch.max.wait.ms** プロパティーおよび **fetch.min.bytes** プロパティーを調整して、より大きなバッチでリクエストとメッセージが配信されるようにすることができます。より高い

値に調整することでスループットが改善されますが、レイテンシーのコストが発生します。生成されるデータ量が少ない場合、より高い値に調整することもできます。

たとえば、**fetch.max.wait.ms** を 500ms に設定し、**fetch.min.bytes** を 16384 バイトに設定した場合、Kafka がコンシューマーからフェッチリクエストを受信すると、いずれかのしきい値に最初に到達した時点でレスポンスが返されます。

逆に、**fetch.max.wait.ms** プロパティーおよび **fetch.min.bytes** プロパティーを調整して、エンドツーエンドのレイテンシーを改善できます。

```
# ...
fetch.max.wait.ms=500 1
fetch.min.bytes=16384 2
# ...
```

1 ブローカーがフェッチリクエストを完了するまで待機する最大時間 (ミリ秒単位)。デフォルトは **500** ミリ秒です。

2 最小バッチサイズ (バイト単位) が使用された場合、最低限到達時にリクエストが送信されます。または、メッセージが **fetch.max.wait.ms** よりも長くキューに入れられると、リクエストが送信されます。遅延を追加すると、メッセージをバッチサイズまで累積できます。

フェッチリクエストサイズの増加によるレイテンシーの短縮

fetch.max.bytes プロパティーおよび **max.partition.fetch.bytes** プロパティーを使用して、Kafka ブローカーからコンシューマーによって取得されるデータの最大量を増やします。

fetch.max.bytes プロパティーは、一度にブローカーから取得されるデータ量の上限をバイト単位で設定します。

max.partition.fetch.bytes は、各パーティションで返されるデータ量の上限をバイト単位で設定します。これは、**max.message.bytes** のブローカーまたはトピック設定に設定されたバイト数よりも大きくする必要があります。

クライアントが消費できるメモリーの最大量は、以下のように概算されます。

> **NUMBER-OF-BROKERS** * fetch.max.bytes and **NUMBER-OF-PARTITIONS** * max.partition.fetch.bytes

メモリー使用量がこれに対応できる場合は、これら 2 つのプロパティーの値を増やすことができます。各リクエストでより多くのデータを許可すると、フェッチリクエストが少なくなるため、レイテンシーが向上されます。

```
# ...
fetch.max.bytes=52428800 1
max.partition.fetch.bytes=1048576 2
# ...
```

1 フェッチリクエストに対して返されるデータの最大量 (バイト単位)。

2 各パーティションに対して返されるデータの最大量 (バイト単位)。

4.6. オフセットをコミットする際のデータ損失または重複の回避

Kafka の **自動コミットメカニズム** により、コンシューマーはメッセージのオフセットを自動的にコミットできます。有効にすると、コンシューマーはブローカーをポーリングして受信したオフセットを 5000ms 間隔でコミットします。

自動コミットのメカニズムは便利ですが、データ損失と重複のリスクが発生します。コンシューマーが多くのメッセージを取得および変換し、自動コミットの実行時にコンシューマーバッファーに処理されたメッセージがある状態でシステムがクラッシュすると、そのデータは失われます。メッセージの処理後、自動コミットの実行前にシステムがクラッシュした場合、リバランス後に別のコンシューマーインスタンスでデータが複製されます。

ブローカーへの次のポーリングの前またはコンシューマーが閉じられる前に、すべてのメッセージが処理された場合は、自動コミットによるデータの損失を回避できます。

データの損失や重複の可能性を最小限に抑えるには、**enable.auto.commit** を **false** に設定し、クライアントアプリケーションを開発して、オフセットのコミットをより詳細に制御できるようにします。または、**auto.commit.interval.ms** を使用してコミットの間隔を減らすことができます。

```
# ...
enable.auto.commit=false (1)
# ...
```

(1) 自動コミットを false に設定すると、オフセットのコミットの制御が強化されます。

enable.auto.commit を **false** に設定すると、**すべての** 処理が実行され、メッセージが消費された後にオフセットをコミットできます。たとえば、Kafka **commitSync** および **commitAsync** コミット API を呼び出すようにアプリケーションを設定できます。

commitSync API は、ポーリングから返されるメッセージバッチのオフセットをコミットします。バッチのメッセージすべての処理が完了したら API を呼び出します。**commitSync** API を使用する場合、アプリケーションはバッチの最後のオフセットがコミットされるまで新しいメッセージをポーリングしません。これがスループットに悪影響する場合は、コミットする頻度が低いか、**commitAsync** API を使用できます。**commitAsync** API はブローカーがコミットリクエストにレスポンスするまで待機しませんが、リバランス時にさらに重複が発生するリスクがあります。一般的な方法として、両方のコミット API をアプリケーションで組み合わせ、コンシューマーをシャットダウンまたはリバランスの直前に **commitSync** API を使用し、最終コミットが正常に実行されるようにします。

4.6.1. トランザクションメッセージの制御

プロデューサー側でトランザクション ID を使用し、冪等性 (**enable.idempotence=true**) を有効にすることを検討してください。これにより、1 回限りの配信を保証します。コンシューマー側で、**isolation.level** プロパティーを使用して、コンシューマーによってトランザクションメッセージが読み取られる方法を制御できます。

isolation.level プロパティーには、有効な 2 つの値があります。

- **read_committed**
- **read_uncommitted** (デフォルト)

read_committed を使用して、コミットされたトランザクションメッセージのみがコンシューマーによって読み取られるようにします。ただし、これによりトランザクションの結果を記録するトランザクションマーカー (**committed** または **aborted**) がブローカーによって書き込まれるまで、コンシューマーはメッセージを返すことができないため、エンドツーエンドのレイテンシーが長くなります。

```
# ...
```

```
enable.auto.commit=false
isolation.level=read_committed (1)
# ...
```

(1) コミットされたメッセージのみがコンシューマーによって読み取られるように、**read_committed** に設定します。

4.7. データ損失を回避するための障害からの復旧

コンシューマーグループ内で障害が発生した場合、Kafka は効果的な検出と回復のために設計されたリバランスプロトコルを提供します。こうした障害による潜在的な影響を最小限に抑えるための重要なストラテジーの 1 つは、**max.poll.records** プロパティーを調整して、効率的な処理とシステムの安定性のバランスを取ることです。このプロパティーは、コンシューマーが 1 回のポーリングでフェッチできるレコードの最大数を決定します。**max.poll.records** を微調整すると、制御された消費率を維持し、コンシューマーによるコンシューマー自体や Kafka ブローカーへの過負荷を防ぐことができます。

さらに、Kafka は、**session.timeout.ms** や **heartbeat.interval.ms** などの高度な設定プロパティーを提供します。これらの設定は通常、より特殊なユースケースのために予約されており、標準的なシナリオでは調整が必要ない場合があります。

session.timeout.ms プロパティーは、コンシューマーがコンシューマーグループ内でアクティブであることを示すために Kafka ブローカーにハートビートを送信しなくてよい最大時間を指定します。コンシューマーがセッションタイムアウト内にハートビートを送信できなかった場合、コンシューマーは非アクティブとみなされます。非アクティブとしてマークされたコンシューマーは、トピックのパーティションのリバランスをトリガーします。**session.timeout.ms** プロパティーの設定値が低すぎると誤検出が発生する可能性があり、設定値が高すぎると障害からの回復が遅れる可能性があります。

heartbeat.interval.ms プロパティーは、コンシューマーが Kafka ブローカーにハートビートを送信する頻度を決定します。連続するハートビート間の間隔が短いため、コンシューマーの障害をより迅速に検出できます。ハートビートの間隔は、セッションタイムアウトより短くする必要があります (通常はセッションタイムアウトの 3 分の 2 にする必要があります)。ハートビートの間隔が短くなると、誤ってリバランスを行う可能性が低くなりますが、ハートビートを頻繁に行うとブローカーリソースのオーバーヘッドが増えます。

4.8. オフセットポリシーの管理

auto.offset.reset プロパティーを使用して、オフセットがコミットされていない場合にコンシューマーの動作を制御するか、コミットされたオフセットが有効でなくなったりします。

コンシューマーアプリケーションを初めてデプロイし、既存のトピックからメッセージを読み取る場合について考えてみましょう。これは **group.id** が初めて使用されるため、**__consumer_offsets** トピックには、このアプリケーションのオフセット情報は含まれません。新しいアプリケーションは、ログの始めからすべての既存メッセージの処理を開始するか、新しいメッセージのみ処理を開始できます。デフォルトのリセット値は、パーティションの最後に開始する **latest** で、一部のメッセージは見逃されることを意味します。データの損失は避けたいが、処理量を増やしたい場合は、**auto.offset.reset** を **earliest** に設定して、パーティションの先頭から開始します。

また、ブローカーに設定されたオフセットの保持期間 (**offsets.retention.minutes**) が終了したときにメッセージが失われるのを防ぐために、**earliest** オプションの使用も検討してください。コンシューマーグループまたはスタンドアロンコンシューマーが非アクティブで、保持期間中にオフセットをコミットしない場合、以前にコミットされたオフセットは **__consumer_offsets** から削除されます。

```
# ...
heartbeat.interval.ms=3000 (1)
```

```
session.timeout.ms=45000 2
auto.offset.reset=earliest 3
# ...
```

1 予想されるリバランスに応じて、ハートビートの間隔を短くして調整します。

2 タイムアウトの期限が切れる前に Kafka ブローカーによってハートビートが受信されなかった場合、コンシューマーはコンシューマーグループから削除され、リバランスが開始されます。ブローカー設定に **group.min.session.timeout.ms** と **group.max.session.timeout.ms** がある場合、セッションタイムアウト値はその範囲内である必要があります。

3 パーティションの最初に戻り、オフセットがコミットされなかった場合にデータの損失を回避するために、**earliest** 値に設定します。

1 つのフェッチリクエストで返されるデータ量が大きい場合、コンシューマーが処理する前にタイムアウトが発生することがあります。この場合、**max.partition.fetch.bytes** を減らしたり、**session.timeout.ms** を増やすこともできます。

4.9. リバランスの影響の最小限に抑える方法

グループ内のアクティブなコンシューマー間のパーティションをリバランスするには、次の操作分の時間がかかります。

- コンシューマーによるオフセットのコミット
- 作成される新しいコンシューマーグループ
- グループリーダーによるグループメンバーへのパーティションの割り当て
- 割り当てを受け取り、取得を開始するグループのコンシューマー

リバランスプロセスにより、サービスのダウンタイムが長くなる可能性があります。特に、コンシューマーグループクラスターのローリング再起動中にリバランスが繰り返し発生する場合に顕著です。

この状況では、グループ内の各コンシューマーインスタンスに一意の識別子 (**group.instance.id**) を割り当てることで、**静的メンバーシップ** を導入できます。静的メンバーシップは永続性を使用し、セッションタイムアウト後の再起動時にコンシューマーインスタンスが認識されるようにします。その結果、コンシューマーはトピックパーティションの割り当てを維持し、障害または再起動後にグループに再度参加する際の不要なリバランスが減ります。

さらに、**max.poll.interval.ms** 設定を調整すると、長時間にわたる処理タスクに起因するリバランスが防止され、新しいメッセージのポーリング間の最大間隔を指定できるようになります。**max.poll.records** プロパティーを使用すると、各ポーリング中にコンシューマーバッファーから返されるレコード数が制限されます。レコード数を減らすと、コンシューマーはより少ないメッセージをより効率的に処理できるようになります。長時間のメッセージ処理が避けられない場合は、そのようなタスクをワーカースレッドのプールにオフロードすることを検討してください。この並列処理アプローチを使用すると、大量のレコードでコンシューマーに負荷がかかることに起因する遅延や潜在的なリバランスが防止されます。

```
# ...
group.instance.id=UNIQUE-ID 1
max.poll.interval.ms=300000 2
max.poll.records=500 3
# ...
```

1. 一意のインスタンス ID により、新しいコンシューマーインスタンスに同じトピックパーティションが割り当てられます。
2. コンシューマーがメッセージの処理を継続していることを確認する間隔を設定します。
3. コンシューマーから返される処理済のレコードの数を設定します。

第5章 KAFKA プロデューサー設定のチューニング

設定プロパティーを使用して、Kafka プロデューサーのパフォーマンスを最適化します。標準の Kafka プロデューサー設定オプションを使用できます。設定を調整してスループットを最大化すると、レイテンシーが増加する可能性があり、その逆も同様です。必要なバランスを取得するために、プロデューサー設定を実験して調整する必要があります。

プロデューサーを設定するときは、パフォーマンスと動作に大きな影響を与えるため、次の点を慎重に考慮してください。

圧縮

メッセージをネットワーク経由で送信する前に圧縮すると、ネットワーク帯域幅が節約され、ディスクストレージ要件を減らすことができます。ただし、圧縮および展開のプロセスにより CPU 使用率が増加するという追加コストがかかります。

バッチ処理

プロデューサーがメッセージを送信するときのバッチサイズと時間間隔を調整すると、スループットとレイテンシーに影響を与える可能性があります。

パーティション設定

Kafka クラスターのパーティショニングストラテジーは、並列処理と負荷分散によりプロデューサーをサポートできます。これにより、プロデューサーは複数のパーティションに同時に書き込むことができ、各パーティションはメッセージのシェアを均等に受け取ります。他のストラテジーには、フォールトトレランスのためのトピックレプリケーションが含まれる場合があります。

アクセスのセキュリティー保護

Kafka へのセキュアなアクセスを管理 するユーザーアカウントを設定して、認証、暗号化、および承認のセキュリティー対策を実装します。

5.1. 基本のプロデューサー設定

接続およびシリアライザープロパティーはすべてのプロデューサーに必要です。通常、追跡用のクライアント ID を追加し、プロデューサーで圧縮してリクエストのバッチサイズを減らすことが推奨されます。

基本的なプロデューサー設定は、以下のようになります。

- パーティション内のメッセージの順序は保証されません。
- ブローカーに到達するメッセージの完了通知は持続性を保証しません。

基本的なプロデューサー設定プロパティー

```
# ...
bootstrap.servers=localhost:9092 1
key.serializer=org.apache.kafka.common.serialization.StringSerializer 2
value.serializer=org.apache.kafka.common.serialization.StringSerializer 3
client.id=my-client 4
compression.type=gzip 5
# ...
```

1 (必須) Kafka ブローカーの **host:port** ブートストラップサーバーアドレスを使用して Kafka クラスターに接続するようプロデューサーを指示します。プロデューサーはアドレスを使用して、クラスター内のすべてのブローカーを検出し、接続します。サーバーがダウンした場合に備えて、コンマ区切りリストを使用して 2 つまたは 3 つのアドレスを指定しますが、クラスター内のすべてのブ

ローカーのリストを提供する必要はありません。

2 (必須) メッセージがブローカーに送信される前に、各メッセージの鍵をバイトに変換するシリアライザー。

3 (必須) メッセージがブローカーに送信される前に、各メッセージの値をバイトに変換するシリアライザー。

4 (オプション) クライアントの論理名。リクエストのソースを特定するためにログおよびメトリックで使用されます。

5 (オプション) メッセージを圧縮するコーデック。これは、送信されます。場合によっては圧縮形式で保存され、コンシューマーに到達すると展開されます。圧縮は、スループットを向上させ、ストレージの負荷を軽減するのに役立ちますが、圧縮または圧縮解除のコストが非常に高くなる可能性がある低レイテンシーのアプリケーションには適していない可能性があります。

5.2. データの持続性

メッセージ配信の確認は、メッセージが失われる可能性を最小限に抑えます。デフォルトでは、acks プロパティーを **acks=all** に設定すると確認が有効になります。プロデューサーがブローカーによる確認を待機する最大時間を制御し、メッセージ送信の潜在的な遅延に対処するには、**delivery.timeout.ms** プロパティーを使用できます。

メッセージ配信の承認

```
# ...
acks=all 1
delivery.timeout.ms=120000 2
# ...
```

1 **acks=all** を指定すると、リーダーレプリカが強制的に、特定数のフォロワーに対するメッセージを複製してから、メッセージリクエストが正常に受信されたことを確認します。

2 送信リクエストの完了まで待機する最大時間 (ミリ秒単位)。この値を **MAX_LONG** に設定すると、Kafka に無限の再試行を委任できます。デフォルトは **120000** または 2 分です。

acks=all 設定は、最も強力な配信保証を提供しますが、プロデューサーがメッセージを送信して確認レスポンスを受け取るまでの待ち時間が長くなります。このような強力な保証が必要ない場合は、**acks=0** または **acks=1** を設定すると、配信が保証されないか、リーダーレプリカがログにレコードを書き込んだことを確認するだけになります。

acks=all を使用すると、リーダーはすべての同期レプリカがメッセージ配信を確認するまで待機します。トピックの **min.insync.replicas** 設定は、同期レプリカの確認レスポンスに必要な最小数を設定します。確認レスポンスの数には、リーダーとフォロワーが含まれます。

一般的に、以下の設定を使用して操作を開始します。

- プロデューサーの設定:
  - **acks=all** (デフォルト)
- トピックレプリケーションのブローカー設定:
  - **default.replication.factor=3** (default = **1**)

- **min.insync.replicas=2** (default = **1**)

トピックの作成時に、デフォルトのレプリケーション係数をオーバーライドできます。また、トピック設定のトピックレベルで **min.insync.replicas** を上書きすることもできます。

Streams for Apache Kafka は、Kafka のマルチノードデプロイメントの例の設定ファイルを使用します。

以下の表は、リーダーレプリカを複製するフォロワーの可用性に応じてこの設定がどのように動作するかを示しています。

表5.1 フォロワーの可用性

利用可能なフォロワーと同期しているフォロワーの数	確認	プロデューサーがメッセージを送信できるか?
2	リーダーは、フォロワー 2 つからの確認レスポンスを待つ	はい
1	リーダーは、フォロワー 1 つからの確認を待つ	はい
0	リーダーが例外を発生させる	いいえ

トピックのレプリケーション係数が 3 の場合は、1 つのリーダーレプリカと 2 つのフォロワーが作成されます。この設定では、1 つのフォロワーが利用できない場合にプロデューサーはそのまま続行できます。in-sync レプリカから障害のあったブローカーを削除している間または、新しいリーダーを作成している間に、遅延が生じる可能性があります。2 つ目のフォロワーも利用できない場合、メッセージ配信は成功しません。リーダーは、メッセージ配信の成功を確認する代わりに、エラー (**not enough replicas**) をプロデューサーに送信します。プロデューサーは同等の例外を発生させます。**retries** 設定を使用すると、プロデューサーは失敗したメッセージリクエストを再送信できます。

注記

システムに障害が発生すると、バッファーの未送信データが失われる可能性があります。

5.3. 順序付き配信

メッセージは 1 度だけ配信されるため、冪等プロデューサーは重複を回避します。障害発生時でも配信の順序が維持されるように、ID とシーケンス番号がメッセージに割り当てられます。データの一貫性を保つために **acks=all** を使用している場合は、順序付けられた配信に冪等性を有効にすることが妥当です。デフォルトでは、プロデューサーに対して冪等性が有効になっています。冪等性を有効にすると、メッセージの順序を維持するために、進行中の同時リクエストの数を最大 5 に設定できます。

冪等を使用した順序付き配信

```
# ...
enable.idempotence=true 1
max.in.flight.requests.per.connection=5 2
```

```
acks=all (3)
retries=2147483647 (4)
# ...
```

1. 冪等プロデューサーを有効にするには **true** に設定します。
2. 冪等配信では、インフライトリクエストの数が 1 を越えることがありますがメッセージの順序は維持されます。デフォルトのインフライトリクエストの数は 5 です。
3. **acks** を **all** に設定します。
4. 失敗したメッセージリクエストを再送信する試行回数を設定します。

パフォーマンスコストのために **acks=all** を使用せず、冪等性を無効にすることを選択した場合は、進行中の (未確認の) リクエストの数を 1 に設定して、順序を維持します。そうしないと、**Message-A** が失敗し、**Message-B** がブローカーに書き込まれた後にのみ成功する可能性があります。

冪等を使用しない順序付け配信

```
# ...
enable.idempotence=false (1)
max.in.flight.requests.per.connection=1 (2)
retries=2147483647
# ...
```

1. **false** に設定すると、冪等プロデューサーを無効にします。
2. インフライトリクエストの数のみを **1** に設定します。

5.4. 信頼性の保証

冪等は、1 つのパーティションへの書き込みを 1 回だけ行う場合に便利です。トランザクションを冪等と使用すると、複数のパーティション全体で 1 度だけ書き込みを行うことができます。

トランザクションは、同じトランザクション ID を使用するメッセージが 1 度作成され、**すべて** がそれぞれのログに書き込まれるか、**何も** 書き込まれないかのどちらかになることを保証します。

```
# ...
enable.idempotence=true
max.in.flight.requests.per.connection=5
acks=all
retries=2147483647
transactional.id=UNIQUE-ID (1)
transaction.timeout.ms=900000 (2)
# ...
```

1. 一意のトランザクション ID を指定します。
2. タイムアウトエラーが返されるまでのトランザクションの最大許容時間 (ミリ秒単位) を設定します。デフォルトは **900000** または 15 分です。

トランザクション保証を維持するには、**transactional.id** の選択が重要です。トランザクション ID は、一意なトピックパーティションセットに使用する必要があります。たとえば、トピックパーティション名からトランザクション ID への外部マッピングを使用したり、競合を回避する関数を使用してトピックパーティション名からトランザクション ID を算出したりすると、これを実現できます。

5.5. プロデューサーのスループットおよびレイテンシーの最適化

通常、システムの要件は、指定のレイテンシー内であるメッセージの割合に対して、特定のスループットのターゲットを達成することです。たとえば、95 % のメッセージが 2 秒以内に完了確認される、1 秒あたり 500,000 個のメッセージをターゲットとします。

プロデューサーのメッセージングセマンティック (メッセージの順序付けと持続性) は、アプリケーションの要件によって定義される可能性があります。たとえば、アプリケーションが提供する重要なプロパティーや保証を壊さずに **acks=0** または **acks=1** を使用するオプションはありません。

ブローカーの再起動は、パーセンタイルの高い統計に大きな影響を与えます。たとえば、長期間では、99% のレイテンシーはブローカーの再起動に関する動作によるものです。これは、ベンチマークを設計したり、本番環境のパフォーマンスで得られた数字を使用してベンチマークを行い、そのパフォーマンスの数字を比較したりする場合に検討する価値があります。

目的に応じて、Kafka はスループットとレイテンシーのプロデューサーパフォーマンスを調整するために多くの設定パラメーターと設定方法を提供します。

メッセージのバッチ処理 (linger.ms および batch.size)

メッセージのバッチ処理では、同じブローカー宛のメッセージをより多く送信するために、メッセージの送信を遅らせ、単一の生成リクエストでバッチ処理できるようにします。バッチ処理では、スループットを増やすためにレイテンシーを長くして妥協します。時間ベースのバッチ処理は **linger.ms** を使用して設定され、サイズベースのバッチ処理は **batch.size** を使用して設定されます。

圧縮 (compression.type)

メッセージ圧縮処理により、プロデューサー (メッセージの圧縮に費やされた CPU 時間) のレイテンシーが追加されますが、リクエスト (および場合によってはディスクの書き込み) を小さくするため、スループットが増加します。圧縮に価値があるかどうか、および使用に最適な圧縮は、送信されるメッセージによって異なります。圧縮は **KafkaProducer.send()** を呼び出すスレッドで発生するため、アプリケーションでこのメソッドのレイテンシーが重要となる場合は、より多くのスレッドの使用を検討する必要があります。

パイプライン処理 (max.in.flight.requests.per.connection)

パイプライン処理は、以前のリクエストへのレスポンスを受け取る前により多くのリクエストを送信します。通常、パイプライン処理を増やすとスループットの向上し、そのしきい値に達すると、バッチ処理の悪化などの他の影響がスループットへの影響を打ち消し始めます。

レイテンシーの短縮

アプリケーションが **KafkaProducer.send()** メソッドを呼び出すと、送信前のメッセージに対して一連の操作が実行されます。

- インターセプション: 設定されたインターセプターによってメッセージが処理されます。
- シリアライゼーション: メッセージが適切な形式にシリアライズされます。
- パーティションの割り当て: 各メッセージが特定のパーティションに割り当てられます。
- 圧縮: ネットワーク帯域幅を節約するためにメッセージが圧縮されます。

- バッチ処理: 圧縮されたメッセージが、パーティション固有のキュー内のバッチに追加されます。

これらの操作の間、**send()** メソッドは一時的にブロックされます。また、**buffer.memory** が満杯の場合、またはメタデータが利用できない場合も、同メソッドはブロックされたままになります。

バッチは、以下のいずれかが行われるまでキューに残ります。

- バッチが満杯になる (**batch.size**による)。
- **linger.ms** によって導入された遅延が経過する。
- 送信者が、他のパーティションのバッチを同じブローカーにディスパッチする準備ができており、このバッチを含めることができる。
- プロデューサーがフラッシュまたは閉じられている。

send() のブロッキング状態がレイテンシーに与える影響を最小限に抑えるには、バッチ処理とバッファリングの設定を最適化します。**linger.ms** プロパティーと **batch.size** プロパティーを使用し、より多くのメッセージを単一の生成リクエストにバッチ処理してスループットを高めます。

```
# ...
linger.ms=100 1
batch.size=16384 2
buffer.memory=33554432 3
# ...
```

1. **linger.ms** プロパティーは、より大きなメッセージのバッチが蓄積され、リクエストで送信されるように、ミリ秒単位の遅延を追加します。デフォルトは **0** です。
2. **batch.size** の最大値をバイト単位で指定した場合、最大値に達したとき、またはメッセージが **linger.ms** を超えてキューに入っていたとき (いずれか早いほう) にリクエストが送信されます。遅延を追加すると、メッセージをバッチサイズまで累積できます。
3. バッファーサイズは、少なくともバッチサイズと同じ大きさである必要があり、バッファリング、圧縮、およびインフライトリクエストに対応できる必要があります。

スループットの増加

デフォルトを置き換えるカスタムパーティショナーを使用して、指定したパーティションにメッセージを送信することで、メッセージリクエストのスループットを向上できます。

```
# ...
partitioner.class=my-custom-partitioner 1

# ...
```

1. カスタムパーティショナーのクラス名を指定します。

第6章 大量のメッセージ処理

Streams for Apache Kafka デプロイメントで大量のメッセージを処理する必要がある場合は、設定オプションを使用してスループットとレイテンシーを最適化できます。

プロデューサーとコンシューマーの設定は、Kafka ブローカーへの要求のサイズと頻度を制御するのに役立ちます。設定オプションの詳細は、以下を参照してください。

- プロデューサー向けの Apache Kafka 設定ドキュメント
- コンシューマー向けの Apache Kafka 設定ドキュメント

また、Kafka Connect ランタイムソースコネクター (MirrorMaker 2 を含む) およびシンクコネクターで使用されるプロデューサーとコンシューマーで同じ設定オプションを使用することもできます。

ソースコネクター

- Kafka Connect ランタイムのプロデューサーは、メッセージを Kafka クラスターに送信します。
- MirrorMaker 2 の場合、ソースシステムは Kafka であるため、コンシューマーはソース Kafka クラスターからメッセージを取得します。

シンクコネクター

- Kafka Connect ランタイムのコンシューマーは、Kafka クラスターからメッセージを取得します。

コンシューマーの場合、1 回のフェッチリクエストでフェッチされるデータの量を増やして、レイテンシーを短縮することができます。**fetch.max.bytes** および **max.partition.fetch.bytes** プロパティーを使用して、フェッチ要求のサイズを増やします。**max.poll.records** プロパティーを使用して、コンシューマーバッファーから返されるメッセージ数の上限を設定することもできます。

MirrorMaker 2 の場合、ソースからメッセージをフェッチする特定のコンシューマーに関連して、ソースコネクターレベル (**consumer.***) で **fetch.max.bytes**、**max.partition.fetch.bytes**、および **max.poll.records** の値を設定します。

プロデューサーの場合は、1 つの生成リクエストで送信されるメッセージバッチのサイズを増やすことができます。**batch.size** プロパティーを使用してバッチサイズを増やします。バッチサイズを大きくすると、送信する準備ができている未処理のメッセージの数と、メッセージキュー内のバックログのサイズが減少します。同じパーティションに送信されるメッセージはまとめてバッチ処理されます。バッチサイズに達すると、プロデュースリクエストがターゲットクラスターに送信されます。バッチサイズを大きくすると、プロデュースリクエストが遅延し、より多くのメッセージがバッチに追加され、同時にブローカーに送信されます。これにより、多数のメッセージを処理するトピックパーティションが複数ある場合に、スループットが向上します。

プロデューサーが適切なプロデューサーバッチサイズに対して処理するレコードの数とサイズを考慮します。

linger.ms を使用してミリ秒単位の待機時間を追加し、プロデューサーの負荷が減少したときにプロデュースリクエストを遅らせます。遅延は、最大バッチサイズ未満の場合に、より多くのレコードをバッチに追加できることを意味します。

ソースコネクターレベル (**producer.override.***) で **batch.size** および **linger.ms** の値を設定します。これは、ターゲット Kafka クラスターにメッセージを送信する特定のプロデューサーに関連するためです。

Kafka Connect ソースコネクターでは、ターゲット Kafka クラスターへのデータストリーミングパイプラインは以下のようになります。

Kafka Connect ソースコネクターのデータストリーミングパイプライン

外部データソース → (Kafka Connect タスク) ソースメッセージキュー → プロデューサーバーッファー → ターゲット Kafka トピック

Kafka Connect シンクコネクターの場合、ターゲット外部データソースへのデータストリーミングパイプラインは次のとおりです。

Kafka Connect シンクコネクターのデータストリーミングパイプライン

ソース Kafka トピック → (Kafka Connect タスク) シンクメッセージキュー → コンシューマーバッファー → 外部データソース

MirrorMaker 2 の場合、ターゲット Kafka クラスターへのデータミラーリングパイプラインは次のとおりです。

MirrorMaker 2 のデータミラーリングパイプライン

ソース Kafka トピック → (Kafka Connect タスク) ソースメッセージキュー → プロデューサーバーッファー → ターゲット Kafka トピック

プロデューサーは、バッファー内のメッセージをターゲット Kafka クラスター内のトピックに送信します。これが発生している間、Kafka Connect タスクは引き続きデータソースをポーリングして、ソースメッセージキューにメッセージを追加します。

ソースコネクターのプロデューサーバーッファーのサイズは、**producer.override.buffer.memory** プロパティーを使用して設定されます。タスクは、バッファーがフラッシュされる前に、指定されたタイムアウト期間 (**offset.flush.timeout.ms**) 待機します。これは、送信されたメッセージがブローカーによって確認され、コミットされたデータがオフセットされるのに十分な時間です。ソースタスクは、シャットダウン中を除き、オフセットをコミットする前にプロデューサーがメッセージキューを空にするのを待ちません。

プロデューサーがソースメッセージキュー内のメッセージのスループットについていけない場合、バッファリングは、**max.block.ms** で制限された期間内にバッファーに使用可能なスペースができるまでブロックされます。バッファー内に未確認のメッセージがあれば、この期間中に送信されます。これらのメッセージが確認されてフラッシュされるまで、新しいメッセージはバッファーに追加されません。

次の設定変更を試して、未処理メッセージの基になるソースメッセージキューを管理可能なサイズに保つことができます。

- **offset.flush.timeout.ms** のデフォルト値 (ミリ秒) を増やす
- 十分な CPU およびメモリーリソースがあることを確認します。
- 以下を実行して、並行して実行されるタスクの数を増やします。
  - **tasksMax** プロパティーを使用して並行して実行するタスクの数を増やす
  - **replicas** プロパティーを使用してタスクを実行するワーカーノードの数の増加

使用可能な CPU とメモリーリソース、およびワーカーノードの数に応じて、並列実行できるタスクの数を検討してください。必要な効果が得られるまで、設定値を調整し続けることを推奨します。

6.1. 大量メッセージ用の KAFKA CONNECT の設定

Kafka Connect は、ソースの外部データシステムからデータをフェッチし、それを Kafka Connect ランタイムプロデューサーに渡して、ターゲットクラスターにレプリケートします。

次の例は、**KafkaConnect** カスタムリソースを使用した Kafka Connect の設定を示しています。

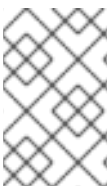
大量のメッセージを処理するための Kafka Connect 設定の例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: "true"
spec:
  replicas: 3
  config:
    offset.flush.timeout.ms: 10000
    # ...
  resources:
    requests:
      cpu: "1"
      memory: 2Gi
    limits:
      cpu: "2"
      memory: 2Gi
  # ...
```

プロデューサー設定は、**KafkaConnector** カスタムリソースを使用して管理されるソースコネクター用に追加されます。

大量のメッセージを処理するためのソースコネクターの設定例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
metadata:
  name: my-source-connector
  labels:
    strimzi.io/cluster: my-connect-cluster
spec:
  class: org.apache.kafka.connect.file.FileStreamSourceConnector
  tasksMax: 2
  config:
    producer.override.batch.size: 327680
    producer.override.linger.ms: 100
    # ...
```



注記

FileStreamSourceConnector および **FileStreamSinkConnector** は、コネクターの例として提供されています。これらを **KafkaConnector** リソースとしてデプロイする方法については、**KafkaConnector** リソース のデプロイ を 参照してください。

シンクコネクターのコンシューマー設定が追加されます。

大量のメッセージを処理するためのシンクコネクターの設定例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
metadata:
  name: my-sink-connector
  labels:
    strimzi.io/cluster: my-connect-cluster
spec:
  class: org.apache.kafka.connect.file.FileStreamSinkConnector
  tasksMax: 2
  config:
    consumer.fetch.max.bytes: 52428800
    consumer.max.partition.fetch.bytes: 1048576
    consumer.max.poll.records: 500
    # ...
```

KafkaConnector カスタムリソースの代わりに Kafka Connect API を使用してコネクターを管理している場合は、コネクター設定を JSON オブジェクトとして追加できます。

大量のメッセージを処理するためのソースコネクター設定を追加するための curl 要求の例

```
curl -X POST \
  http://my-connect-cluster-connect-api:8083/connectors \
  -H 'Content-Type: application/json' \
  -d '{ "name": "my-source-connector",
    "config":
    {
      "connector.class":"org.apache.kafka.connect.file.FileStreamSourceConnector",
      "file": "/opt/kafka/LICENSE",
      "topic":"my-topic",
      "tasksMax": "4",
      "type": "source"
      "producer.override.batch.size": 327680
      "producer.override.linger.ms": 100
    }
}'
```

6.2. 大量のメッセージ用の MIRRORMAKER 2 の設定

MirrorMaker 2 はソースクラスターからデータを取得し、それを Kafka Connect ランタイムプロデューサーに渡して、ターゲットクラスターにレプリケーションします。

次の例は、**KafkaMirrorMaker2** カスタムリソースを使用した MirrorMaker 2 の設定を示しています。

大量のメッセージを処理するための MirrorMaker 2 設定の例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaMirrorMaker2
metadata:
  name: my-mirror-maker2
spec:
  version: 3.7.0
  replicas: 1
```

```
  connectCluster: "my-cluster-target"
  clusters:
  - alias: "my-cluster-source"
    bootstrapServers: my-cluster-source-kafka-bootstrap:9092
  - alias: "my-cluster-target"
    config:
      offset.flush.timeout.ms: 10000
    bootstrapServers: my-cluster-target-kafka-bootstrap:9092
  mirrors:
  - sourceCluster: "my-cluster-source"
    targetCluster: "my-cluster-target"
    sourceConnector:
      tasksMax: 2
      config:
        producer.override.batch.size: 327680
        producer.override.linger.ms: 100
        consumer.fetch.max.bytes: 52428800
        consumer.max.partition.fetch.bytes: 1048576
        consumer.max.poll.records: 500
    # ...
  resources:
    requests:
      cpu: "1"
      memory: Gi
    limits:
      cpu: "2"
      memory: 4Gi
```

6.3. MIRRORMAKER 2 メッセージフローの確認

Prometheus と Grafana を使用してデプロイメントを監視している場合は、MirrorMaker 2 のメッセージフローを確認できます。

Streams for Apache Kafka で提供される MirrorMaker 2 Grafana ダッシュボードの例は、フラッシュパイプラインに関連する次のメトリックを示しています。

- Kafka Connect の未処理メッセージキューにあるメッセージの数
- プロデューサーバーッファーの使用可能なバイト数
- オフセットコミットタイムアウト (ミリ秒)

これらのメトリックを使用して、メッセージの量に基づいて設定を調整する必要があるかどうかを判断できます。

関連情報

- メトリックの概要
- Kafka Connect コネクターの追加

第7章 大きなメッセージサイズの処理

Kafka のデフォルトのメッセージバッチサイズは 1 MB で、ほとんどのユースケースで最大のスループットを得るのに最適です。Kafka は、十分なディスク容量があれば、スループットを下げてより大きなバッチに対応できます。

サイズの大きいメッセージは、次の 4 つの方法で処理できます。

1. ブローカー、プロデューサー、およびコンシューマーは、サイズのより大きいメッセージに対応するように設定されています。
2. プロデューサー側のメッセージ圧縮 が、圧縮メッセージをログに書き込みます。
3. 参照ベースのメッセージングでは、メッセージのペイロード内の他のシステムに保存されているデータへの参照のみが送信されます。
4. インラインメッセージングが、同じキーを使用するチャンクにメッセージを分割し、これらを Kafka Streams などのストリームプロセッサーを使用して、出力に組み合わせます。

非常に大きなメッセージを処理するのでない限り、この設定アプローチが推奨されます。参照ベースのメッセージングとメッセージ圧縮のオプションは、他のほとんどの状況に対応します。いずれのオプションを使用する場合も、パフォーマンスの問題が発生しないように注意する必要があります。

7.1. より大きなメッセージを処理するための KAFKA コンポーネントの設定

サイズの大きいメッセージはシステムのパフォーマンスに影響を与え、メッセージ処理を複雑化させる可能性があります。サイズの大きいメッセージを回避できない場合は、設定オプションを利用できます。サイズの大きいメッセージを効率的に処理し、メッセージフローのブロックを防ぐには、次の設定を調整することを検討してください。

- 最大レコードバッチサイズの調整:
  - すべてのトピックでより大きなレコードバッチサイズをサポートするには、ブローカーレベルで **message.max.bytes** を設定します。
  - 個々のトピックのより大きなレコードバッチサイズをサポートするには、トピックレベルで **max.message.bytes** を設定します。
- 各パーティションフォロワーによってフェッチされるメッセージの最大サイズを増やします (**replica.fetch.max.bytes**)。
- プロデューサーのバッチサイズ (**batch.size**) を増やして、単一のプロデュース要求で送信されるメッセージバッチのサイズを増やします。
- より大きなレコードバッチに対応するために、プロデューサー (**max.request.size**) とコンシューマー (**fetch.max.bytes**) の最大リクエストサイズを大きく設定します。
- 各パーティションでコンシューマーに返されるデータの量に、より高い上限 (**max.partition.fetch.bytes**) を設定します。

最大レコードバッチサイズに対応できるように、バッチ要求の最大サイズが少なくとも **message.max.bytes** と同じ大きさであることを確認します。

ブローカー設定の例

```
message.max.bytes: 10000000
replica.fetch.max.bytes: 10485760
```

プロデューサーの設定例

```
batch.size: 327680
max.request.size: 10000000
```

コンシューマー設定の例

```
fetch.max.bytes: 10000000
max.partition.fetch.bytes: 10485760
```

また、Kafka Bridge、Kafka Connect、MirrorMaker 2 などの他の Kafka コンポーネントで使用されるプロデューサーとコンシューマーを設定して、サイズがより大きなメッセージをより効率的に処理することもできます。

Kafka Bridge

特定のプロデューサーおよびコンシューマー設定プロパティーを使用して Kafka Bridge を設定します。

- プロデューサー用の **producer.config**
- コンシューマー向け **consumer.config**

Kafka Bridge の設定例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaBridge
metadata:
  name: my-bridge
spec:
  # ...
  producer:
    config:
      batch.size: 327680
      max.request.size: 10000000
  consumer:
    config:
      fetch.max.bytes: 10000000
      max.partition.fetch.bytes: 10485760
      # ...
```

Kafka Connect

Kafka Connect の場合、プロデューサーとコンシューマーの設定プロパティーの接頭辞を使用して、メッセージの送受信を行うソースコネクターとシンクコネクターを設定します。

- ソースコネクターが Kafka クラスターにメッセージを送信するために使用するプロデューサーの **producer.override**
- シンクコネクターが Kafka クラスターからメッセージを取得するために使用するコンシューマーの **consumer**

Kafka Connect ソースコネクター設定の例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
metadata:
  name: my-source-connector
  labels:
    strimzi.io/cluster: my-connect-cluster
spec:
  # ...
  config:
    producer.override.batch.size: 327680
    producer.override.max.request.size: 10000000
    # ...
```

Kafka Connect シンクコネクター設定の例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
metadata:
  name: my-sink-connector
  labels:
    strimzi.io/cluster: my-connect-cluster
spec:
  # ...
  config:
    consumer.fetch.max.bytes: 10000000
    consumer.max.partition.fetch.bytes: 10485760
    # ...
```

MirrorMaker 2

MirrorMaker 2 の場合、プロデューサーとコンシューマーの設定プロパティーの接頭辞を使用して、ソース Kafka クラスターからメッセージを取得するソースコネクターを設定します。

- ターゲットの Kafka クラスターにデータを複製するために使用されるランタイム Kafka Connect プロデューサーの **producer.override**
- シンクコネクターがソース Kafka クラスターからメッセージを取得するために使用するコンシューマーの **consumer**

MirrorMaker 2 ソースコネクター設定の例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaMirrorMaker2
metadata:
  name: my-mirror-maker2
spec:
  # ...
  mirrors:
  - sourceCluster: "my-cluster-source"
    targetCluster: "my-cluster-target"
    sourceConnector:
      tasksMax: 2
      config:
```

```
producer.override.batch.size: 327680
producer.override.max.request.size: 10000000
consumer.fetch.max.bytes: 10000000
consumer.max.partition.fetch.bytes: 10485760
# ...
```

7.2. プロデューサー側の圧縮

プロデューサー設定の場合は、Gzip などの **compression.type** を指定します。これは、プロデューサーによって生成されたデータのバッチに適用されます。ブローカー設定の **compression.type=producer** (デフォルト) を使用すると、ブローカーはプロデューサーが使用した圧縮を保持します。プロデューサーとトピックの圧縮が一致しない場合は常に、ブローカーはバッチをログに追加する前に再度圧縮する必要があります。これはブローカーのパフォーマンスに影響を与えます。

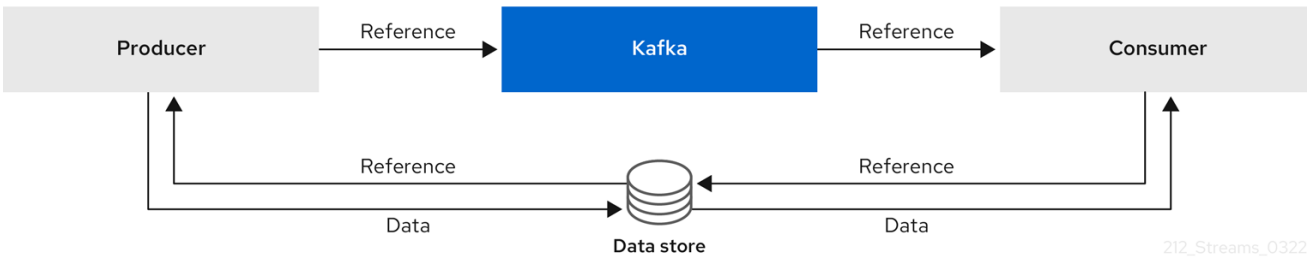
圧縮はまた、プロデューサーに追加の処理オーバーヘッドを追加し、コンシューマーに解凍オーバーヘッドを追加しますが、バッチにより多くのデータが含まれるため、メッセージデータが適切に圧縮される場合、スループットに役立つことがよくあります。

プロデューサー側の圧縮とバッチサイズの微調整を組み合わせて、最適なスループットを促進します。メトリックを使用すると、必要な平均バッチサイズの測定に役立ちます。

7.3. 参照ベースのメッセージング

参照ベースのメッセージングは、メッセージの大きさがわからない場合のデータ複製に役立ちます。この設定が機能するには、外部データストアは高速で永続性があり、高可用性である必要があります。データはデータストアに書き込まれ、データへの参照が返されます。プロデューサーは、Kafka への参照が含まれるメッセージを送信します。コンシューマーはメッセージから参照を取得し、これを使用してデータストアからデータを取得します。

7.4. 参照ベースのメッセージングフロー



メッセージを渡すにはより多くの通信が必要なため、エンドツーエンドのレイテンシーが増加します。このアプローチのもう 1 つの重大な欠点は、Kafka メッセージがクリーンアップされたときに、外部システムのデータが自動的にクリーンアップされないことです。ハイブリッドアプローチは、大きなメッセージのみをデータストアに送信し、標準サイズのメッセージを直接処理することです。

インラインメッセージング

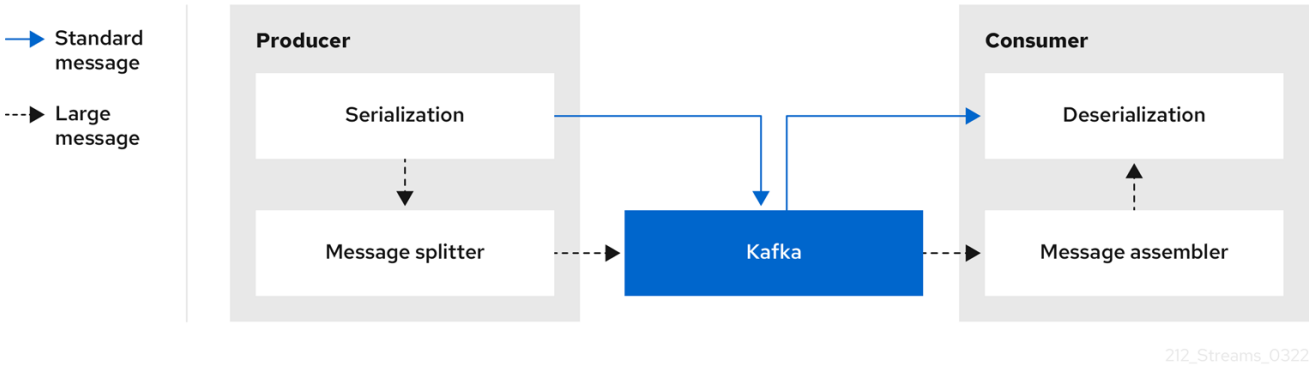
インラインメッセージングは複雑ですが、参照ベースのメッセージングのように外部システムに依存するオーバーヘッドはありません。

メッセージが大きすぎる場合、生成するクライアントアプリケーションは、データをシリアライズしてからチャンクにする必要があります。その後、プロデューサーは Kafka **ByteArraySerializer** を使用

し、送信前に各チャンクを再度シリアライズするのと同様のものを使用します。コンシューマーはメッセージを追跡し、完全なメッセージが得られるまでチャンクをバッファリングします。消費側のクライアントアプリケーションは、デシリアライズの前にアセンブルされたチャンクを受け取ります。完全なメッセージは、チャンクになったメッセージの各セットの最初または最後のチャンクのオフセットに従って、消費する残りのアプリケーションに配信されます。

メッセージ配信の正確な追跡を保証し、再バランス調整中の重複を防ぐために、コンシューマーはすべてのメッセージチャンクを受信して処理した後にのみオフセットをコミットする必要があります。チャンクは複数のセグメントに分散される可能性があります。コンシューマー側の処理では、セグメントが後で削除された場合にチャンクが使用できなくなる可能性を考慮する必要があります。

図7.1 インラインメッセージングフロー



インラインメッセージングは、特に一連の大きなメッセージを並行して処理する場合に必要なバッファリングのために、コンシューマー側でパフォーマンスのオーバーヘッドが発生します。大きなメッセージのチャンクはインターリーブされる可能性があるため、バッファー内の別の大きなメッセージのチャンクが不完全な場合、メッセージのすべてのチャンクが消費されたときにコミットできるとは限りません。このため、バッファリングは通常、メッセージチャンクを永続化するか、コミットロジックを実装することでサポートされます。

付録A サブスクリプションの使用

Streams for Apache Kafka は、ソフトウェアサブスクリプションから提供されます。サブスクリプションを管理するには、Red Hat カスタマーポータルでアカウントにアクセスします。

アカウントへのアクセス

1. access.redhat.com に移動します。
2. アカウントがない場合は作成します。
3. アカウントにログインします。

サブスクリプションのアクティベート

1. access.redhat.com に移動します。
2. **My Subscriptions** に移動します。
3. **Activate a subscription** に移動し、16 桁のアクティベーション番号を入力します。

Zip および Tar ファイルのダウンロード

zip または tar ファイルにアクセスするには、カスタマーポータルを使用して、ダウンロードする関連ファイルを検索します。RPM パッケージを使用している場合、この手順は必要ありません。

1. ブラウザーを開き、access.redhat.com/downloads で Red Hat カスタマーポータルの **Product Downloads** ページにログインします。
2. **Streams for Apache Kafka** エントリーの場所を **INTEGRATION AND AUTOMATION** カテゴリーで特定します。
3. 必要な Streams for Apache Kafka 製品を選択します。**Software Downloads** ページが開きます。
4. コンポーネントの **Download** リンクをクリックします。

DNF を使用したパッケージのインストール

パッケージとすべてのパッケージ依存関係をインストールするには、以下を使用します。

```
dnf install <package_name>
```

ローカルディレクトリーからダウンロード済みのパッケージをインストールするには、以下を使用します。

```
dnf install <path_to_download_package>
```

改訂日時: 2024-04-30