Red Hat AMQ Streams 2.3

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: 2023-04-10

Red Hat AMQ Streams 2.3 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

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 KAFKA チューニングの概要

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パフォーマンスを最適化します。OCP および RHEL で AMQ Streams の設定プロパティーを指定できます。

最小セットの設定プロパティーが必要ですが、プロパティーを追加または調整して、プロデューサーとコンシューマーが Kafka ブローカーと対話する方法を変更できます。たとえば、クライアントがリアルタイムでデータにレスポンスできるように、メッセージのレイテンシーおよびスループットをチューニングできます。

メトリクスを分析して初期設定を行う場所を判断することから始め、必要な設定になるまで段階的に変更を加え、さらにメトリクスの比較を行うことができます。

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。

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 AMQ Streams 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。RHEL の AMQ Streams 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
```

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。

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 AMQ Streams の設定
- RHEL での AMQ Streams の使用

第2章 マネージドブローカーの設定

AMQ Streams を OpenShift にデプロイするときは、**Kafka** カスタムリソースの **config** プロパティーを介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、AMQ Streams によって直接管理されます。

そのため、OpenShift で AMQ Streams を使用している場合は、次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。

除外項目の一覧については、**KafkaClusterSpec** スキーマ参照 を参照してください。

RHEL で AMQ Streams を使用する場合、これらの除外は適用されません。この場合は、基本的なブローカー設定にこれらのプロパティーを追加して、ブローカーを識別し、安全なアクセスを提供する必要があります。

RHEL での AMQ Streams 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
- RHEL での Kafka の設定

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。AMQ Streams 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お勧めします。

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 AMQ Streams 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. データ保持ポリシーでのログの管理

Kafka はログを使用してメッセージデータを保存します。ログは、さまざまなインデックスに関連付けられた一連のセグメントです。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。セグメントは、コンシューマーからのフェッチリクエストに対応するときに読み取られます。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。一度にアクティブにできるセグメントは1つだけです。古いセグメントは、削除対象となるまで保持されます。

ブローカーレベルでの設定では、ログセグメントの最大サイズをバイト単位で設定し、アクティブなセグメントがロールされるまでの時間をミリ秒単位で設定します。

```
# ...
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上書きできます。これらの値を下げるまたは上げる必要があるかどうかは、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。セグメントも削除の対象となる頻度が少なくなります。

時間ベースまたはサイズベースのログの保持およびクリーンアップポリシーを設定して、ログを管理しやすくすることができます。要件によっては、ログ保持の設定を使用して古いセグメントを削除できます。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、ディスク領域が超過しないように、ログに必要なストレージ領域がバインドされます。

期間ベースのログの保持には、時間、分、およびミリ秒に基づいて保持期間を設定します。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。

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=1680000
# ...
```

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。ディスクの使用状況は常に監視する必要がありますが、-1 の設定は、ディスクがいっぱいになると問題が発生する可能性があり、修正が難しいため、一般的にはお勧めしません。

サイズベースのログの保持には、最大ログサイズ (ログのすべてのセグメント) をバイト単位で設定します。

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

つまり、通常、ログが定常状態に達すると、およそ **log.retention.bytes /log.segment.bytes** のセグメントが含まれます。最大ログサイズに達すると、古いセグメントが削除されます。

最大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前に遅延を追加する場合は、トピック設定の特定のトピックについて、ブローカーレベルまたは **file.delete.delay.ms** のトピックで **log.segment.delete.delay.ms** を使用して遅延を追加できます。

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

3.7. クリーンアップポリシーによるログデータの削除

古いログデータを削除する方法は、**ログクリーナー** 設定によって決定されます。

ログクリーナーは、ブローカーに対してデフォルトで有効になっています。

```
# ...
log.cleaner.enable=true
# ...
```

ログ圧縮クリーンアップポリシーを使用している場合は、ログクリーナーを有効にする必要があります。クリーンアップポリシーは、トピックまたはブローカーレベルで設定できます。ブローカーレベルの設定は、ポリシーが設定されていないトピックのデフォルトです。

ログの削除、ログの圧縮、その両方を行うためにポリシーを設定できます。

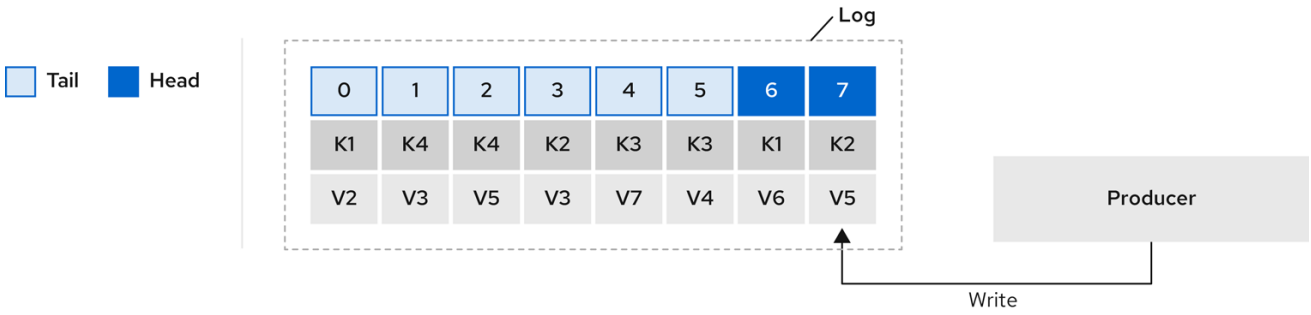
```
# ...
log.cleanup.policy=compact,delete
# ...
```

delete ポリシーは、データ保持ポリシーを使用したログの管理に対応します。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。**compact** ポリシーは、各メッセージキーの最新のメッセージを維持することを保証します。ログコンパクションは、メッセージ値の変更が可能で、最新の更新を保持する場合に適しています。

ログを削除するようにクリーンアップポリシーが設定されている場合、ログの保持制限に基づいて古いセグメントが削除されます。それ以外の場合、ログクリーナーが有効になっておらず、ログの保持制限がないと、ログは増え続けます。

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、同じキーを持つ別のレコードがログの後半で発生した場合、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。キーを使用していない場合、関連するメッセージを識別するためにキーが必要になるため、コンパクションを使用することはできません。Kafka は、各キーの最新のメッセージが保持されることを保証しますが、圧縮されたログ全体に重複が含まれないことを保証するものではありません。

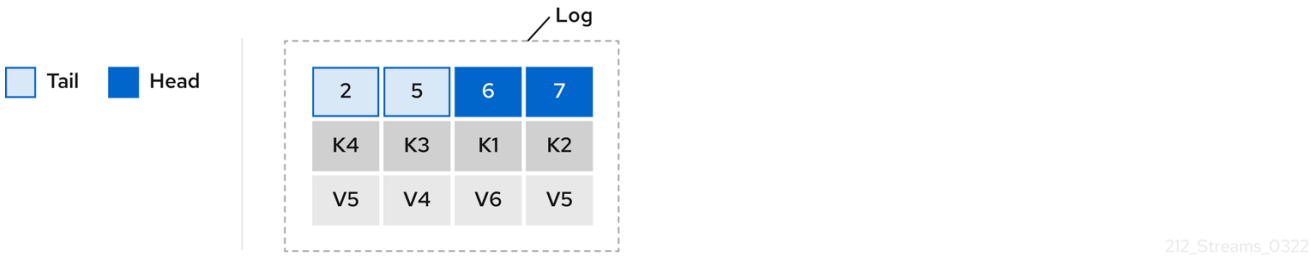
図3.1 コンパクション前のオフセットの位置によるキー値の書き込みを示すログ



鍵を使用してメッセージを特定することで、Kafka のコンパクションは特定のメッセージキーの最新メッセージ (オフセットが最大) を維持し、最終的に同じキーを持つ以前のメッセージを破棄します。つまり、最新状態のメッセージは常に利用可能であり、その特定のメッセージの古いレコードは、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。

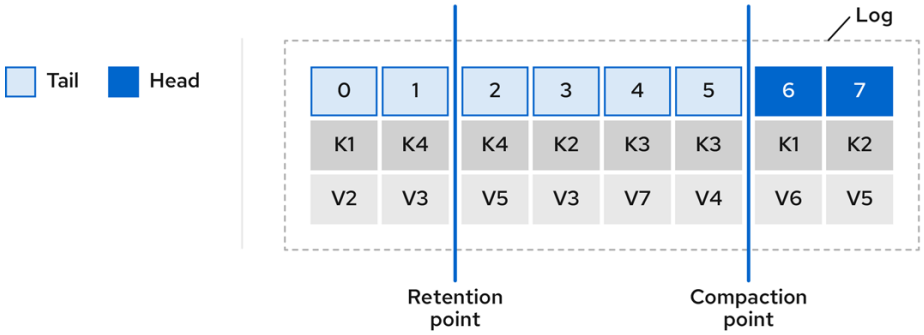
図3.2 コンパクション後のログ



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。この場合、ログの圧縮 **お**

よび 削除を行うためにポリシーを設定します。コンパクションおよび削除を選択した場合、まずログデータが圧縮され、ログの先頭にあるキーでレコードが削除されます。その後、ログ保持しきい値より前のデータは削除されます。

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



ログのクリーンアップがチェックされ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、トピック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。

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
# ...
```

古いログデータの削除を選択した場合、パージする前に削除されたデータを保持する期間をミリ秒単位で設定できます。

```
# ...
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。廃棄 (tombstone) には null 値があり、値が削除されることを示すマーカーとして機能します。コンパクション後に廃棄 (tombstone) のみが保持されます。これは、コンシューマーがメッセージが削除されたことを認識するのに十分な期間である必要があります。古いメッセージが削除され、値がないと、tombstone キーもパーティションから削除されます。

3.8. ディスク使用率の管理

ログクリーンアップに関する他の設定には数多くありますが、特に重要なのはメモリー割り当てです。

重複排除 (deduplication) プロパティーは、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の合計メモリーを指定します。バッファー負荷係数で使用されるメモリーの割合の上限を設定できます。

```
# ...
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
# ...
```

3.9. 大きなメッセージサイズの処理

メッセージのデフォルトのバッチサイズは 1MB 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。

大きなメッセージサイズは、以下の 4 つの方法で処理されます。

1. 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。
2. 参照ベースのメッセージングが、メッセージの値で他のシステムに格納されているデータへの参照のみを送信します。
3. 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。
4. ブローカーおよびプロデューサー/コンシューマークライアントアプリケーションが、より大きなメッセージサイズを処理するように構築されます。

リファレンスベースのメッセージングおよびメッセージ圧縮オプションが推奨されます。これはほとんどの状況に対応します。これらのオプションのいずれかを使用する場合は、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。

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 を使用すると、ブローカーは使用されるプロデューサーを圧縮します。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。

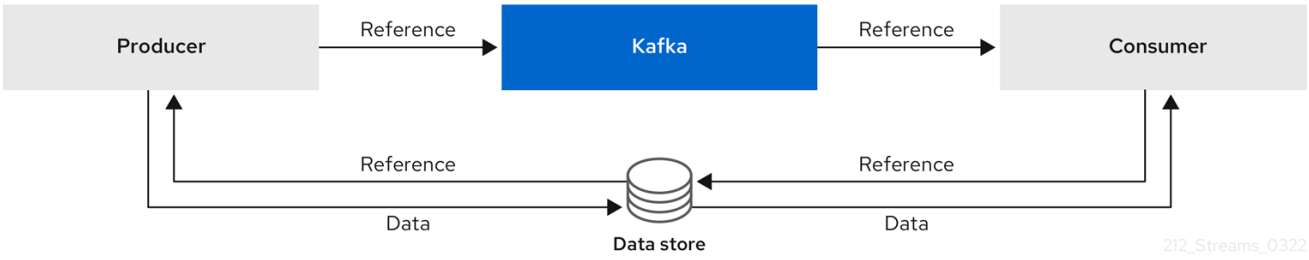
圧縮はまた、プロデューサーに追加の処理オーバーヘッドを追加し、コンシューマーに解凍オーバーヘッドを追加しますが、バッチにより多くのデータが含まれるため、メッセージデータが適切に圧縮される場合、スループットに役立つことがよくあります。

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。メトリクスを使用すると、必要な平均バッチサイズの測定に役立ちます。

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。

図3.4 参照ベースのメッセージングフロー



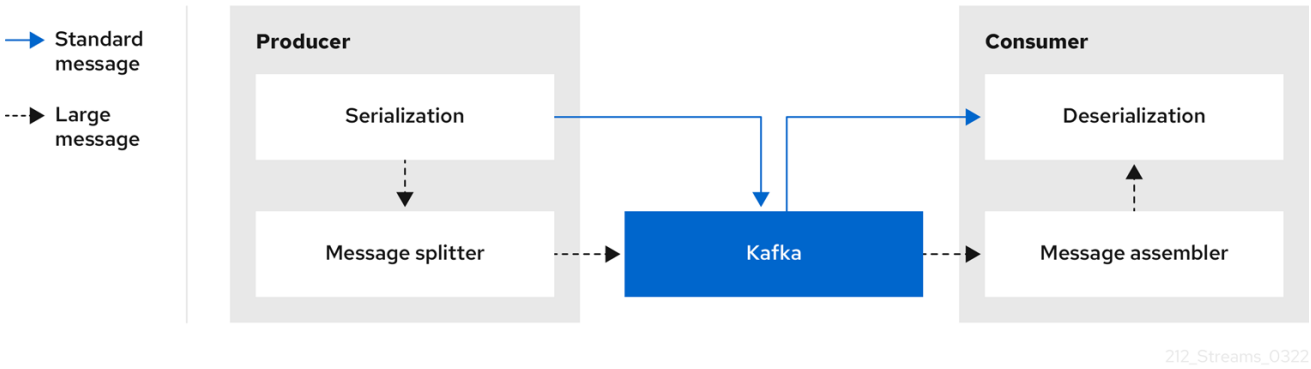
メッセージを渡すにはより多くの通信が必要なため、エンドツーエンドのレイテンシーが増加します。このアプローチのもう 1 つの重大な欠点は、Kafka メッセージがクリーンアップされたときに、外部システムのデータが自動的にクリーンアップされないことです。ハイブリッドアプローチは、大きなメッセージのみをデータストアに送信し、標準サイズのメッセージを直接処理することです。

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ーリングします。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。リバランス中の重複を避けるために、完全なメッセージの正常な配信がオフセットメタデータと照合されます。

図3.5 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ーリングのために、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、バッファーリングは通常、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。

より大きなメッセージを処理するための設定

より大きなメッセージを回避できない場合、およびメッセージフローの任意の時点でブロックを回避するために、メッセージ制限を増やすことができます。これを行うには、トピックレベルで **message.max.bytes** を設定し、個別のトピックの最大レコードバッチサイズを設定します。ブローカーレベルで **message.max.bytes** を設定すると、すべてのトピックに大きなメッセージが許可されます。

ブローカーは、**message.max.bytes** で設定された制限よりも大きなメッセージを拒否します。プロデューサー (**max.request.size**) およびコンシューマー (**message.max.bytes**) のバッファーサイズは、より大きなメッセージに対応できなければなりません。

3.10. 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お勧めします。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。

3.11. 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一覧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。

AMQ Streams の Cruise Control 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。

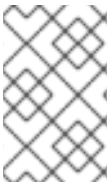
Cruise Control で検出された割り当てが実際にバランスが取れている場合には、優先リーダーがパーティションのリーダーとなる必要があります。Kafka は、優先リーダーが使用されていることを自動的に確認し (可能な場合)、必要に応じて現在のリーダーを変更します。これにより、クラスターは CruiseControl が検出した時のバランスの取れた状態に保たれます。

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

AMQ Streams で提供される Grafana ダッシュボードでは、複製の数が最低数未満のパーティションや、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。

3.12. 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。

3.13. 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

特定のユースケースに合わせて調整されたオプションのプロパティーとともに、基本的なコンシューマー設定を使用します。

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。

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クスで使用されます。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。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. 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。

4.4. 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)。

4.5. 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。

4.5.1. 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。

4.6. データ損失を回避するための障害からの復旧

session.timeout.ms および **heartbeat.interval.ms** プロパティーを使用して、コンシューマーグループ内のコンシューマー障害をチェックし、復旧にかかる時間を設定します。

session.timeout.ms プロパティーは、コンシューマーグループ内のコンシューマーがブローカーとのコンタクトを絶った場合に、非アクティブとみなされ、グループ内のアクティブなコンシューマー間で **リバランス** が発生するまでの最大時間をミリ秒単位で指定します。グループのリバランス時に、パーティションはグループのメンバーに再割り当てされます。

heartbeat.interval.ms プロパティーは、コンシューマーがアクティブで接続されていることを示す、コンシューマーグループコーディネーターへの **ハートビート** チェックの間隔をミリ秒単位で指定します。通常、ハートビートの間隔はセッションタイムアウトの間隔の 3 分の 2 にする必要があります。

session.timeout.ms プロパティーを低く設定すると、失敗したコンシューマーが先に検出され、リバランスがより迅速に実行されます。ただし、タイムアウトの値を低くしすぎて、ブローカーがハートビートを時間内に受信できず、不必要なリバランスがトリガーされることがないように気を付けてください。

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。

4.7. 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。

4.8. 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、次の操作時間分、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

このプロセスは明らかに、サービスのダウンタイムを増加させます。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繰り返し発生する場合に顕著です。

このような場合、**静的メンバーシップ** の概念を使用してリバランスの数を減らすことができます。リバランスによって、コンシューマーグループメンバー全体でトピックパーティションが割り当てられます。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。

コンシューマーグループコーディネーターは、**group.instance.id** プロパティーを使用して指定される一意の ID を使用して新しいコンシューマーインスタンスを特定できます。再起動時には、コンシューマーには新しいメンバー ID が割り当てられますが、静的メンバーとして、同じインスタンス ID を使用し、同じトピックパーティションの割り当てが行われます。

コンシューマーアプリケーションが少なくとも **max.poll.interval.ms** ミリ秒毎にポーリングへの呼び出しを行わない場合、コンシューマーが失敗したと見なされ、リバランスが発生します。アプリケーションがポーリングから返されたすべてのレコードを時間内に処理できない場合は、**max.poll.interval.ms** プロパティーを使用してコンシューマーから新しいメッセージのポーリングの間隔をミリ秒単位で指定して、リバランスを回避することができます。または、**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レコード数の上限を設定できます。これにより、アプリケーションが **max.poll.interval.ms** の制限内で、処理するレコード数を少なくできます。

```
# ...
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

特定のユースケースに合わせて調整されたオプションのプロパティーとともに、基本的なプロデューサー設定を使用します。

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。

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クスで使用されます。
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。**acks** プロパティーが **acks=all** に設定されている場合、確認レスポンスはデフォルトで有効になっています。

メッセージ配信の承認

```
# ...
acks=all (1)
# ...
```

(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。

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。

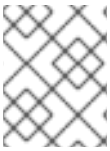
AMQ Streams は、Kafka のマルチノードデプロイメントの例の設定ファイルを使用します。

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。

冪等を使った順序付き配信

```
# ...
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っ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()** を呼び出す場合、メッセージは以下のようになります。

- インターセプターによる処理
- シリアライズ
- パーティションへの割り当て
- 圧縮処理
- パーティションごとのキュー内のメッセージのバッチに追加

ここで、**send()** メソッドが返されます。そのため、**send()** がブロックされる時間は、以下によって決定されます。

- インターセプター、シリアライザー、およびパーティショナーで費やされた時間
- 使用される圧縮アルゴリズム
- 圧縮に使用するバッファーの待機に費やされた時間

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
- 送信者は他のパーティションのメッセージバッチを同じブローカーに送信しようとし、このバッチの追加も可能である
- プロデューサーがフラッシュまたは閉じられている

バッチ処理とバッファーの設定を参照して、レイテンシーをブロックする **send()** の影響を軽減します。

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

1. **linger** 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、バッファー、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

メッセージの配信および送信リクエストの完了までの最大待機時間を調整して、メッセージリクエストのスループットを向上します。

また、カスタムパーティションを作成してデフォルトを置き換えることで、メッセージを指定のパーティションに転送することもできます。

```
# ...
delivery.timeout.ms=120000 1
partitioner.class=my-custom-partitioner 2

# ...
```

1. 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。

2. カスタムパーティショナーのクラス名を指定します。

付録A サブスクリプションの使用

AMQ Streams 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は、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. **INTEGRATION AND AUTOMATION** カテゴリーで、**AMQ Streams for Apache Kafka**エントリーを見つけます。
3. 必要な AMQ Streams 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: 2023-04-06