記事
Toshihiko Minamoto · 2021年3月15日 16m read

InterSystems IRIS のグローバルを使ったトランザクション。

InterSystems IRIS では、情報を格納する「グローバル」というユニークなデータ構造をサポートしています。 基本的に、グローバルとは、マルチレベルのインデックスを持つ永続配列であり、トランザクションの実行やツリー構造のスピーディなトラバーサルといった機能が備えられているほか、ObjectScript として知られるプログラミング言語にも対応しています。

ここから先、少なくともコードサンプルについては、グローバルの基礎を理解されているという想定のもとに話しを進めていきます。

グローバルはデータを保存するための魔法の剣です パート1
グローバルはデータを保存するための魔法の剣ですパート2 - ツリー
グローバルはデータを保存するための魔法の剣です パート3 - 疎な配列

グローバルは、普通のテーブルとは全く異なる構造でデータを格納し、OSI モデルの下位層で動作します。
それでは、グローバルを使ったトランザクションとはいかなるもので、どのような特性が見られるのでしょうか。

リレーショナルデータベースの理論では、ACID テスト (Wikipedia で ACID を参照する)
に合格するトランザクションこそが、適切に実装されたトランザクションとされています。

- *不可分性*: トランザクションで発生する変更内容がすべて記録されるか、そのいずれも記録されない。
  Wikipedia で不可分性 (データベースシステム) を見る。
- *一貫性*: トランザクションが完了した後、データベース内部の論理状態が一貫している。
  いろいろな意味で、この要件はプログラマーに適用されるものですが、SQL
  データベースの場合では、外部キーにも適用されます。
- *分離性*: 並列に実行されるトランザクションの結果はお互いを影響しない。
- *永続性 (あるいは持続性)*: トランザクションが完了した後に、OSI モデルの物理層の問題 (電源障害など)
  が発生しても、トランザクションで変更されたデータには影響しない。

グローバルは、非リレーショナルなデータ構造です。 メモリーフットプリントを最小限に抑えながらハードウェアの超高速作業をサポートできるようにデザインされています。 それでは、IRIS/docker-image
を使って、トランザクションがグローバルに実装される仕組みを見てみましょう。

1. 不可分性

3 の値を一緒にデータベースに保存する必要があるが、そうならない場合はそのいずれも保存されないという状況について考えます。

不可分性を一番手っ取り早く確認するには、ターミナルで次のコードを入力します。

```
Kill ^a
TSTART
Set ^a(1) = 1
Set ^a(2) = 2
Set ^a(3) = 3
TCOMMIT
```

最後に次を入力します。

```
ZWRITE ^a
```

結果は以下のようになるはずです。

```
^a(1)=1
^a(2)=2
^a(3)=3
```

予想とおり、不可分性を確認できました。 では、わざとエラーを導入してタスクを少し複雑にします。トランザクションが部分的に保存されるのか、全く保存されないのかを見てみましょう。
まずは、先ほどと同じように不可分性を確認します。

```
Kill ^a
TSTART
Set ^a(1) = 1
Set ^a(2) = 2
Set ^a(3) = 3
```

今回は、ここで docker kill my-iris というコマンドを使い、コンテナを強制的に終了します。これにより SIGKILL (即座にプロセスを停止する) シグナルが送られるため、電源を強制的にオフにしているようなものです。
コンテナを再起動してから、グローバルの中身をチェックして、結果を確認します。
トランザクションは部分的に保存されているでしょうか？

```
ZWRITE ^a

???????????
```

いいえ、何も保存されませんでした。 これにより、アクシデントでサーバーが停止する場合、IRIS データベースでは不可分性が保証されることが分かりました。

では、変更内容を意図的にキャンセルするとしたらどうでしょう？ 次のように rollback コマンドを使って試してみます。

```
Kill ^a

TSTART
Set ^a(1) = 1
Set ^a(2) = 2
Set ^a(3) = 3
TROLLBACK 1

ZWRITE ^a

Nothing got out
```

またしても、何も保存されませんでした。

2. 永続性

グローバルは、リレーショナルテーブルよりも低い層でデータを格納する構造であることを覚えていますか？また、グローバルデータベースでは、インデックスもグローバルとして格納されます。 従い、永続性の要件を満たすには、グローバルノードの値が変更される同じトランザクションにインデックの変更も含める必要があります。

例えば、^person
というグローバルがあり、それに個人情報を格納するとします。キーにはソーシャルセキュリティ番号 (SSN) を使用します。

```
^person(1234567, 'firstname') = 'Sergey'
^person(1234567, ‘lastname’) = ‘Kamenev’
^person(1234567, ‘phone’) = ‘+74995555555
...
```

以下のように、^index というキーを作成し、ラストネームだけ、もしくはファーストネームとラストネームの組み合わせを使って、素早く検索できるようにしました。

```
^index(‘Kamenev’, ‘Sergey’, 1234567) = 1
```

データベースの永続性を維持するには、person を以下のように追加する必要があります。

```
TSTART
^person(1234567, ‘firstname’) = ‘Sergey’
^person(1234567, ‘lastname’) = ‘Kamenev’
^person(1234567, ‘phone’) = ‘+74995555555
^index(‘Kamenev’, ‘Sergey’, 1234567) = 1
TCOMMIT
```

従い、person を削除する場合は、以下のトランザクションを使う必要があります。

```
TSTART
Kill ^person(1234567)
Kill ^index(‘Kamenev’, ‘Sergey’, 1234567)
TCOMMIT
```

つまり、グローバルのような下位層の格納形式を使う場合、どのようなロジックを使ってアプリケーションの永続性の要件を満たすかは、プログラマー次第ということです。

幸い、IRIS
にはトランザクションを整理して、アプリケーションの永続性を保証するためのコマンドが備えられています。
SQL が使用される場合、IRIS は内部でこれらのコマンドを実行して、INSERT、UPDATE、DELETE
式が実行されるときに、基となるグローバルのデータ構造を持続します。
もちろん、トランザクションを開始および停止する SQL コマンドは、IRIS SQL にも備えられているので、(SQL) アプリケーションロジックに活用することができます。

3. 分離性

ここからが本番です。
多くのユーザーが同時に同じデータベースにアクセスして、同じデータを変更するとしましょう。 多くのデベロ

ッパーが同じコードリポジトリにアクセスし、多くのファイルの変更内容を同時にコミットしようと試みる状況に似ています。

データベースはすべてをリアルタイムに処理しなくてはいけません。
大手企業なら通常はバージョンコントロール (ブランチのマージや競合解決の管理など) の担当者がいて、かつ データベースはこれをリアルタイムに処理する必要があることを考えると、その問題の複雑さや、データベースとその機能を支えるコードを適切に設計することが大切なのは言うまでもありません。

データベースは、ユーザーの操作が意味することを理解できなければ、ユーザーが同じデータを操作する際に起こる競合を回避することもできません。 他のトランザクションと矛盾するトランザクションを 1
つキャンセルするか、それらを順番に実行することしかできません。

また、トランザクションの実行中 (コミットの前)
は、データベースの状態の一貫性が失われている能性もあります。
一貫性を失ったデータベースの状態に他のトランザクションがアクセスできるのはよくありません。 しかし、リレーショナルデータベースでは、スナップショットを作成したり、複数バージョンの行を使用したりするなど、アクセスできてしまう方法がたくさんあります。

複数のトランザクションが並列に実行される場合は、お互いを干渉しないことが大切です。
それを確保するのが分離性です。

SQL では、分離性は分離する程度の順に 4 つのレベルで定義されます。 以下がその 4 つのレベルです。

- READ UNCOMMITTED
- READ COMMITTED
- REPEATABLE READ
- SERIALIZABLE

では、それぞれのレベルを一つずつ見ていきましょう。
各レベルの実装コストは、スタックを上がるにつれてほぼ指数関数的に増加していきますので、ご注意ください。

READ UNCOMMITTED は、分離の一番低いレベルですが、一番速いレベルでもあります。
トランザクションは、別のトランザクションによりコミットされた変更内容を読み取ることができます。

READ COMMITTED は、次の分離レベルで、譲歩することを意味します。 トランザクションは、コミットの前にお互いの変更を読み取ることはできませんが、コミットの後ならどの変更でも読み取ることができます。

長時間に渡って実行されるトランザクション (T1) があり、その間に T1
と同じデータを操作するトランザクション T2、T3 ... Tn がそれぞれでコミットしたとします。
このような場合は、T1 のデータをリクエストする度に異なる結果が得られる可能性があります。 これは Non-Repeatable Read (非再現リード) と呼ばれています。

次の分離レベルは、**REPEATABLE READ** です。データの読み取りをリクエストすると、その都度結果のスナップショットがとられるため、ここではもはや非再現リードは起りません。 同じトランザクションの実行中に同じデータがもう一度リクエストされることがあれば、スナップショットが使用されます。
但し、この分離レベルでは、ファンタムデータ
(並列に実行されていた別のトランザクションによってコミットされた新しい文字列)
が読み取られる可能性があります。

一番高い分離レベルは、**SERIALIZABLE** です。 トランザクションで使用 (読み取りまたは変更) されたデータに他のトランザクションがアクセスできるのは、最初のトランザクションが終了した後に限定されるという特徴があります。

まずは、トランザクションのあるスレッドとないスレッドとの間で操作が分離されているかどうかを確認してみましょう。 ターミナルを 2 つ開いて、以下を入力します。

```
Kill ^t Write ^t(1) 2                    TSTART Set ^t(1)=2
```

分離はされていません。
片方のスレッドはトランザクションを開くと、もう片方のスレッドの動作を見ることができます。

それでは、異なるスレッドで実行中のトランザクションがお互いの中の状況を見れるかどうかを確認しましょう。
ターミナルウィンドウを 2 つ開き、2 つのトランザクションを並列に実行します。

Kill ^t TSTART Write ^t(1) 2 TSTART Set ^t(1)=2
画面に A 3 が表示されます。 ここでは、一番シンプルで一番速い分離レベル READ UNCOMMTTED
が設定されています。

原則として、グローバルのような常にスピードを優先する下位層のデータ表現手段では、このレベルが主流となっています。 IRIS SQL では、トランザクションの異なる分離レベルを選択できますが、直接グローバルを操作する時にもっと高い分離レベルが必要になったらどうすればよいでしょう？

ここで、分離レベルの目的とそれぞれの仕組みについて考える必要があります。
例えば、低い分離レベルは、データベースを高速化するための譲歩を目的にしています。

最高の分離レベルを提供する SERIALIZABLE では、並列に実行されたトランザクションの結果がそれらを順に実行した場合の結果と同じになることが保証されます。 これにより、衝突を完全に防ぐことができます。
これは、ObjectScript で適切にロックを使用しても実現できる上に、適用方法も多数あります。 つまり、LOCK コマンドを使えば、普通のロックやインクリメンタルロックを作成したり、ロックを複数個作成することもできます。

それでは、ロックを使って異なる分離レベルを作る方法を見てみましょう。 ObjectScript では、LOCK オペレーターを使います。 このオペレーターは、データを変更するのに必要な排他ロックに限らず、共有ロックと呼ばれるロックも許可します。 共有ロックは、複数のスレッドがデータを読み取る目的で同時にアクセスできるロックであり、そのデータが読み取りプロセス中に他のプロセスによって変更されることはありません。

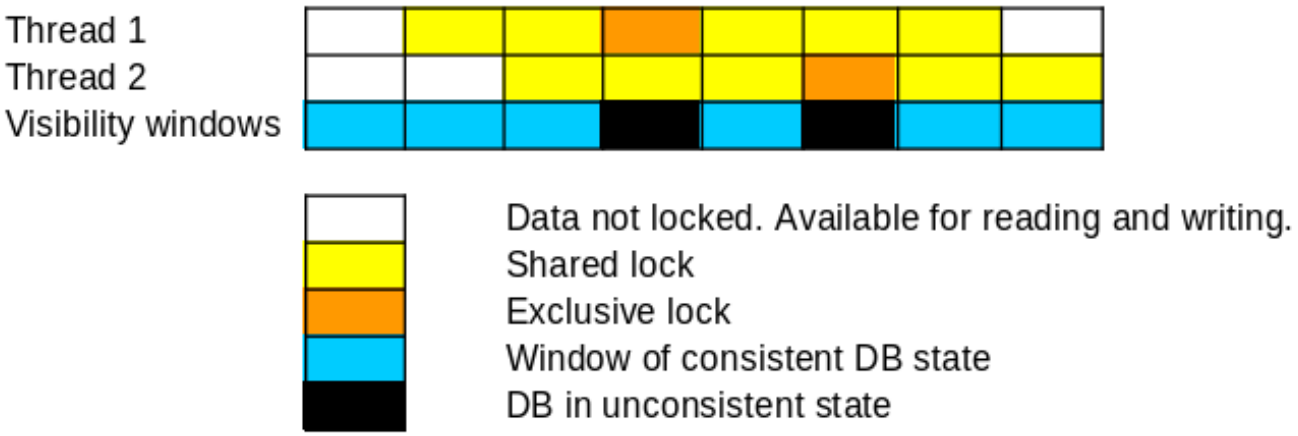
ロック方式に関する詳細については、「ロックと並行処理の制御」と題した記事をお読みください。
ツーフェーズロック方式の詳細については、Wikipedia で「ツーフェーズロック」と題された記事をお読みください。

難しいのは、トランザクションの実行中に、データの状態の一貫性が失われ、そのデータが他のプロセスに表示されてしまうという点です。 これはどうすれば回避できるのか？
この例では、ロックを使って、データベースの状態が一貫して持続される可視性ウィンドウを作成します。
可視性ウィンドウには、ロックを使ってアクセスします。

同じデータに使われているロックは、再利用できるほか、複数のプロセスによる取得が可能です。
これらのロックを使うことで、データが他のプロセスにより変更されるのを防ぎます。
つまり、データベースの状態を一貫させるためのウィンドウを形成するのに使用されるというわけです。

一方の排他ロックは、データを変更するときに使用されるもので、一度に 1 つのプロセスしか取得できません。

排他ロック方式を使用できるシナリオは、2 つあります。 1 つ目は、対象となるデータに対し他のプロセスがロックを取得していないため、どのプロセスでもアクセスできるという場合。 2 つ目は、対象となるデータに対し共有ロックを取得し、かつ排他ロックを最初にリクエストしたプロセスだけがそのデータにアクセスできるという場合。



可視性ウィンドウが狭ければ狭いほど、他のプロセスが待機する時間が長くなる一方で、その中にあるデータベースの状態の一貫性は高まります。

READ COMMITTED では、他のスレッドがコミットしたデータしか見えないようになります。
他のトランザクションのデータがまだコミットされていない場合は、古いバージョンが表示されます。
このおかげで、ロックがリリースされるのを待たずに、作業を並列化することができます。

IRIS では、データの古いバージョンを見るには、特殊な裏技が必要になるので、ロックで何とか対処するしかありません。
共有ロックを使って、一貫性が維持されているポイントでのみデータの読み取りを許可する必要があります。

例えば、複数のユーザー「^person」で構成されるデータベースがあり、ユーザー間で資金の送金が行われるとします。 以下のコードは、person 123 から person 242 に資金が送金されるポイントを示しています。

```
LOCK +^person(123), +^person(242)
TSTART
Set ^person(123, amount) = ^person(123, amount) - amount
Set ^person(242, amount) = ^person(242, amount) + amount
TCOMMIT
LOCK -^person(123), -^person(242)
```

金額が差し引かれる前の段階で、person 123 に送金のリクエストが出るポイントでは、(デフォルトで)
排他ロックが取得されている必要があります。

```
LOCK +^person(123)
Write ^person(123)
```

ですが、ユーザーの個人アカウントのアカウント状況を表示する必要がある場合は、共有ロックを使うか、ロックを一切使わないという選択肢があります。

```
LOCK +^person(123)#"S"
Write ^person(123)
LOCK -^person(123)#"S"
```

但し、データベースの操作がほぼ瞬間的に実行されることを許可するのであれば (グローバルはリレーショナルテ

ーブルよりもずっと下位のレベルの構造であることをお忘れなく)、より高い分離レベルが優先されるのでこのレベルはさほど必要ではなくなります。

以下は、READ COMMITTED の完全な例です。

```
LOCK +^person(123)#”S”, +^person(242)#”S”

Read data (?oncurrent committed transactions can change the data)

LOCK +^person(123), +^person(242)
TSTART
Set ^person(123, amount) = ^person(123, amount) - amount
Set ^person(242, amount) = ^person(242, amount) + amount
TCOMMIT
LOCK -^person(123), -^person(242)

Read data (?oncurrent committed transactions can change the data)

LOCK -^person(123)#”S”, -^person(242)#”S”
```

REPEATABLE READ は、2 番目に高い分離レベルです。 このレベルでは、1 つのトランザクションでデータが複数回読み取られ、その都度同じ結果が出ること、かつ並列に実行されているトランザクションがそのデータを変更できることを許可します。

分離レベルを確実に REPEATABLE READ
にするには、データに対し排他ロックを取得します。それにより、分離レベルは自動的に SERIALIZABLE
に引き上げられます。

```
LOCK +^person(123, amount)

read ^person(123, amount)

?????? (???????? ^person(123, amount) ????????????????)

change ^person(123, amount)

read ^person(123, amount)

LOCK -^person(123, amount)
```

ロックがコンマで区切られている場合は、連続的に取得されていることを意味します。
ですが、以下のようにリストアップされている場合は、一斉にアトミックに取得されます。

```
LOCK +(^person(123),^person(242))
```

SERIALIZABLE は、最も高い分離レベルであり、コストも一番高くなります。 上の例で行ったように、従来のロックを操作する場合は、同じデータを持つすべてのトランザクションが連続的に実行されるようにロックを設定する必要があります。 このアプローチでは、ほぼすべてのロックが排他ロックである必要があり、パフォーマンスを確保するためにグローバルの最も小さいフィールドに対し取得される必要もあります。

^person グローバルから金額を差し引くという場合、SERIALIZABLE 以外のレベルは使えません。 資金の消費は、あくまで連続的なアクションである必要があり、そうでなければ同じ金額が複数回も消費されることになります。

4. 永続性

docker kill my-iris コマンドを使い、コンテナの hard cut-off をテストしたところ、
データベースはこれらのテストに対して良い結果を出し、 問題は一切見られませんでした。

グローバルとロックを管理するためのツール

IRIS Management ポータルでご紹介している以下のツールがお役に立つかもしれません:

ロックを管理。

グローバル。

結論

InterSystems IRIS は、グローバルを使ったアトミックかつ永続的なトランザクションをサポートしています。 データベースとグローバルの一貫性を確保するには、ある程度のプログラミングやトランザクションが必要です。それは、外部キーなどの複雑な構造が組み込まれていないためです。

ロックを取得せずにグローバルを使うのは、READ UNCOMMITTED
の分離レベルを使うのと同じことですが、ロックを使えばそのレベルを SERIALIZABLE
に引き上げることができます。 グローバルを使って実現可能な正確性およびトランザクションの処理速度は、プログラマーのスキルと意図によって大きく左右します。
データを読み取るときに、共有ロックが使用される幅が広ければ広いほど、それだけ分離レベルは高くなります。
また、排他ロックが使用される幅が狭ければ狭いほど、それだけトランザクションの処理速度も速くなります。

#グローバル #データベース #データベースのトランザクション処理 #Caché #InterSystems IRIS

ソースURL:https://jp.community.intersystems.com/post/intersystems-iris-%E3%81%AE%E3%82%B0%E3%83%AD%E3%83%BC%E3%83%90%E3%83%AB%E3%82%92%E4%BD%BF%E3%81%A3%E3%81%9F%E3%83%88%E3%83%A9%E3%83%B3%E3%82%B6%E3%82%AF%E3%82%B7%E3%83%A7%E3%83%B3%E3%80%82