Guia de boas práticas de transações – SQL Server e .NET

Preparado por: Felipe Pessoto, Fernando Henrique Inocêncio Borba Ferreira, Renata Festa, Thaís Domingues,

Data: 04/11/2016

As informações contidas neste documento representam a visão atual da Microsoft Corporation sobre os temas discutidos na data da publicação. Como a Microsoft deve responder às mudanças das condições de mercado, este documento não deve ser interpretado como um compromisso da parte da Microsoft, e a Microsoft não pode assegurar a exatidão de qualquer informação apresentada após a data da publicação.

A MICROSOFT NÃO OFERECE GARANTIAS, EXPRESSAS, IMPLÍCITAS OU ESTATUTÁRIAS, QUANTO ÀS INFORMAÇÕES NESTE DOCUMENTO.

Índice

1 Sumário Executivo

A SEFAZ-SP tem verificado que existem muitos problemas recorrentes relacionados à falta de conhecimento de como que funciona transação, gerando lentidão em alguns sistemas.

Com isso, foi solicitado este guia, contendo informações sobre o que é transação, como utilizar e qual é a melhor prática para utilização.

É recomendado que o mesmo sirva de base para o desenvolvimento de sistemas com qualidade, evitando o uso de certas práticas que levam a um mau desempenho do mesmo.

Este manual se aplica a todas as atuais versões de SQL Server.

2 O que é transação?

Transação é uma sequência de operações que são executadas como uma unidade lógica única de trabalho. Esta unidade tem que ter quatro propriedades para ser tratada como transação. Essas propriedades são chamadas de ACID (Atomicity, Consistency, Isolation, and Durability).

A seguir, detalhes sobre cada uma delas:

- **Atomicidade (Atomicity):** A transação precisa ter uma unidade atômica de trabalho; ou todas as transformações de dados são realizadas ou nenhuma é realizada.
- **Consistência (Consistency):** Após finalizada, uma transação deve deixar todos os dados consistentes. Em um banco relacional, todas as estruturas de dados, como B-tree indexes, precisam estar corretos e consistentes no final da transação.
- **Isolamento (Isolation):** Modificações feitas por transações simultâneas devem ser isoladas de modificações feitas por quaisquer outras transações. Uma transação ou vê dados no estado que estava antes de outra transação simultânea modificar, ou vê os dados depois que a segunda transação foi concluída, mas não vê um estado intermediário.
- **Durabilidade (Durability):** Depois que uma transação foi concluída, seus efeitos são permanentes na aplicação. As alterações persistem mesmo em caso de falha do sistema.

2.1 Atomicidade (Atomicity):

A atomicidade é a propriedade que define que uma transação deve ser uma única unidade de trabalho, ou seja, ou todas as modificações de dados são executadas, ou nenhuma delas é executada.

Dentro de uma transação, o conjunto de operações executado não pode ser dividido ou executado parcialmente, ou seja, para a efetivação de uma transação, todas as operações devem ser executadas com sucesso, caso qualquer operação falhe, toda a transação deve ser desfeita.

2.2 Consistência (Consistency):

Uma transação deve levar uma base de dados de um estado consistente para outro estado consistente, respeitando as regras de integridade dos dados.

Se qualquer operação dentro de uma transação violar alguma regra de consistência da base de dados, toda a transação deve ser desfeita.

2.3 Isolamento (Isolation):

O isolamento é uma das quatro propriedades (ACID) que uma unidade lógica de trabalho deve exibir para se qualificar como uma transação.

Isolamento é a capacidade de proteger as transações dos efeitos das alterações realizadas por outras transações simultâneas. O nível de isolamento é na verdade personalizável para cada transação e permite aos programadores negociarem um maior número de acessos simultâneo em troca de um

maior risco de problemas de integridade de dados, visto que dependendo do nível de isolamento, é possível ler dados sujos. Esse conceito será discutido mais tarde.

Os níveis de isolamento a seguir oferecem um maior isolamento, comparado com o anterior. No entanto, faz isso mantendo bloqueios mais restritivos por períodos mais longos. Os níveis de isolamento de transação são:

- READ UNCOMMITTED
- READ COMMITTED (Padrão no SQL Server)
- REPEATABLE READ
- SERIALIZABLE
- SNAPSHOT*

2.3.1 READ UNCOMMITTED

Implementa leitura suja, que significa que o SQL não realiza nenhum lock de leitura (SHARED) e nenhum lock de gravação (EXCLUSIVE) é respeitado.

Quando esta opção está ligada, os dados que estão sendo lidos podem ser modificados, novas linhas podem ser inseridas ou removidas antes de uma transação ter sido finalizada.

Este nível de isolamento é o menos restritivo e ele tem o mesmo efeito do NOLOCK.

2.3.2 READ COMMITTED

Este nível de isolamento coloca locks de leitura (SHARED) enquanto um dado está sendo lido. Isto evita leitura suja, ou seja, lê apenas dados válidos que já tenham sido commitados. Porém ele permite que o dado seja alterado, inserido novos dados ou deletados antes do fim da transação, fazendo com que você não garanta que você possa ler 2 vezes a mesma informação (sem alteração) dentro de uma transação (nonrepeatable reads) e também podemos ver novos dados que não existiam na primeira leitura (phantom data)

Este nível de isolamento é o padrão do SQL Server

2.3.3 REPEATABLE READ

Neste nível de isolamento os dados tem os registros travados com lock de leitura até o fim da transação. Prevenindo que os dados sejam alterados no meio da transação, porém ainda podemos ver os novos dados que não existiam na primeira leitura (phantom data)

Como o nível de isolamento é maior, ou seja, evita a concorrência entre os diversos processos, o ideal é **utilizar apenas quando necessário**

2.3.4 SERIALIZABLE

Coloca locks em um range de dados prevenindo que dados sejam atualizados ou novos dados sejam inseridos dentro do range que foi lido até o fim da transação. Ex se estou lendo dados entre 2014 até 2015, nenhum dado pode ser alterado / inserido neste range. Porém ainda posso inserir dados em 2016

Este é o nível de isolamento mais restritivo, **utilizar apenas quando necessário**.

2.3.5 SNAPSHOT

Neste nível qualquer comando irá se basear na última versão consistente do dado que existe ao início da transação. Qualquer dado modificado feito após o início não será visível.

Com SNAPSHOT transações de leitura não bloqueiam transações que estão querendo gravar novos dados. E transações que estão gravando ou alterando dados não bloqueiam leitura dos dados.

É possível configurar este nível de isolamento para todo o banco de dados:

```
ALTER DATABASE <DatabaseName>
SET ALLOW_SNAPSHOT_ISOLATION ON
GO
```

Formatado: Português (Brasil)

2.3.6 Lock Modes

O SQL Server usa o usa diferentes modos de lock em recursos, determinando como os objetos podem ser acessados por transações concorrentes.

Estes são os modos de lock que o SQL Server utiliza:

Lock mode	Description
Shared (S)	Usando para operações de leitura, não altera ou atualizada dados, comandos como o SELECT.
Update (U)	Impede um deadlock comum de acontecer, por exemplo, quando multiplas sessões estão lendo, bloqueando e provavelmente atualizando os recursos depois.
Exclusive (X)	Usado para operações de alteração de dados, como INSERT, UPDATE, ou DELETE. Garante que várias atualizações não serão feitas no mesmo recurso ao mesmo tempo.
Intent	Utilizado para estabelecer uma hierarquia de lock. Os tipos de lock são: intent shared (IS), intent exclusive (IX), e shared com intent exclusive (SIX).

2.3.7 Lock Matrix

Requested mode	IS	S	U	IX	SIX	X
Intent shared (IS)	Yes	Yes	Yes	Yes	Yes	No

Shared (S)	Yes	Yes	Yes	No	No	No
Update (U)	Yes	Yes	No	No	No	No
Intent exclusive (IX)	Yes	No	No	Yes	No	No
Shared with intent exclusive (SIX)	Yes	No	No	No	No	No
Exclusive (X)	No	No	No	No	No	No

- **Transaction-SQL**

É possível setar o nível de isolamento por transação no SQL Server:

```
SET TRANSACTION ISOLATION LEVEL
    { READ UNCOMMITTED
    | READ COMMITTED
    | REPEATABLE READ
    | SNAPSHOT
    | SERIALIZABLE
    }
[ ; ]
```

- **ADO**

É possível configurar o nível de isolamento pelo objeto de conexão:

ADO IsolationLevel property	ANSI SQL isolation level
adXactReadUncommitted	READ UNCOMMITTED
adXactReadCommitted	READ COMMITTED
adXactRepeatableRead	REPEATABLE READ
adXactSerializable	SERIALIZABLE

Exemplo:

```
currentIsoLevel = connection.IsolationLevel

connection.IsolationLevel = newIsoLevel
```

- **OLE DB**

Especifica o modo de isolamento para cada transação, para transações paralelas o modo de isolamento será o mesmo para todas as transações.

ISOLEVEL	Description
ISOLATIONLEVEL_READUNCOMMITTED	Read Uncommitted.

ISOLATIONLEVEL_READCOMMITTED	Read Committed.
ISOLATIONLEVEL_REPEATABLEREAD	Repeatable Read.
ISOLATIONLEVEL_SERIALIZABLE	Serializable.

Sintaxe:

```
HRESULT StartTransaction (
   ISOLEVEL                isoLevel,
   ULONG                   isoFlags,
   ITransactionOptions    *pOtherOptions,

  ULONG                   *pulTransactionLevel);
```

- **ODBC**

Especifica o modo de isolamento de transação por conexão odbc:

ValuePtr	Description
SQL_TXN_READ_COMMITTED	Read committed
SQL_TXN_READ_UNCOMMITTED	Read uncommitted
SQL_TXN_REPEATABLE_READ	Repeatable read
SQL_TXN_SERIALIZABLE	Serializable
SQL_TXN_SS_SNAPSHOT	Snapshot

Sintaxe:

```
SQLRETURN SQLSetConnectAttr(
     SQLHDBC       ConnectionHandle,
     SQLINTEGER    Attribute,
     SQLPOINTER    ValuePtr,

     SQLINTEGER    StringLength);
```

2.4 Durabilidade (Durability):

Uma transação deve ser durável, ou seja, uma vez efetivada com sucesso (commit), as modificações efetuadas não serão perdidas mesmo em caso de falhas.

No SQL Server, as transações são comitadas apenas quando persistidas no Transaction Log.

3 Como utilizar dentro de uma procedure?

3.1 Tipos de transação

- **Implicita**

  Por padrão sempre que alguns comandos são executados como por exemplo o INSERT, DELETE ou UPDATE aquele comando representa uma transação. Por exemplo:

  - Se você está inserindo 10.000 linhas e ocorrer um erro na linha 9.999, ele irá fazer rollback de todas linhas que você inseriu.
  - Se você conseguir inserir as 10.000 com sucesso ele irá auto comitar, ou seja, não é necessário executar manualmente COMMIT TRAN. (Observação importante para quem estava acostumado com outros Gerenciadores de banco de dados)

  Apesar de **não recomendado**, este padrão pode ser modificado com o comando "`SET IMPLICIT_TRANSACTIONS ON`" (https://msdn.microsoft.com/en-us/library/ms187807.aspx)

  Com esta opção qualquer comando entre os abaixo irá abrir uma transação que precisará ser commitada.

ALTER TABLE	FETCH	REVOKE
BEGIN TRANSACTION	GRANT	SELECT (See exception below.)
CREATE	INSERT	TRUNCATE TABLE
DELETE	OPEN	UPDATE
DROP		

Exemplo:

```
SET IMPLICIT_TRANSACTIONS ON;

CREATE TABLE #TESTE (x int);

-- No BEGIN TRAN needed here.
INSERT INTO #TESTE VALUES (4);
GO

PRINT N'Tran count in implicit transaction = ' +
  CAST(@@TRANCOUNT AS NVARCHAR(10));

COMMIT TRANSACTION;

PRINT N'Tran count after implicit transaction = '
  + CAST(@@TRANCOUNT AS NVARCHAR(10));
GO
```

```
(1 row(s) affected)

Tran count in implicit
transaction = 1

Tran count after implicit
transaction = 0
```

- **Explicita**

  Quando você necessita referenciar que um ponto onde os dados são referenciados por uma conexão são logicamente e fisicamente consistentes é necessário usar uma transação explicita, por exemplo utilizando "`BEGIN TRANSACTION`"

  Se erros são encontrados, TODAS modificações feitas após o begin transaction serão desfeitas "**ROLLBACK**", retornando todos dados para o último estado consistente do dado.

  Cada transação só é finalizada quando todos comando executaram sem erros e é executado um "`COMMIT TRANSACTION`", fazendo com que as modificações sejam permanentes.

  Exemplo:

```
CREATE TABLE #TESTE (x int);

BEGIN TRANSACTION

       INSERT INTO #TESTE VALUES (4);

ROLLBACK TRAN


BEGIN TRANSACTION

       INSERT INTO #TESTE VALUES (5);

COMMIT TRANSACTION;

SELECT X FROM #TESTE
```

```
(1 row(s) affected)


(1 row(s) affected)


X
-----------
5

(1 row(s) affected)
```

3.2 XACT_ABORT, @@TRANCOUNT e XACT_STATE

Para você controlar o comportamento da transação ou sua **consistência** será necessário **entender como o SQL se comporta para garantir que o seu dado fique consistente ao final da transação**

Por exemplo, quando o XACT_ABORT está ligado, se uma transação recebe um erro de run-time, toda transação é terminada e é feito o ROLLBACK. Quando o XACT_ABORT está desligado [**padrão**], o SQL faz o rollback apenas no comando que gerou erro e continua executando

Você também pode usar esta variável global (@@TRANCOUN) ou a função (XACT_STATE) para saber se uma conexão tem alguma transação aberta e qual o estado dela.

Segue abaixo um exemplo para fixar a ideia

```
SET XACT_ABORT OFF; -- DEFAULT

CREATE TABLE #TEMP (X INT)
INSERT INTO #TEMP VALUES (1)

SELECT
   [@@TRANCOUNT] = @@TRANCOUNT --0 transações
  ,[XACT_STATE] = XACT_STATE()
      --Status 0 = Nenhuma transação ativa

BEGIN TRAN
 BEGIN TRAN
  SELECT
    [@@TRANCOUNT] = @@TRANCOUNT --2 transações
   ,[XACT_STATE] = XACT_STATE()
      --Status 1 = Transação ativa

  INSERT INTO #TEMP VALUES (2) -- OK
  BEGIN TRY
   INSERT INTO #TEMP VALUES (3) -- OK
   INSERT INTO #TEMP VALUES (1/0) -- ERROR
  END TRY
  BEGIN CATCH
   SELECT
     [@@TRANCOUNT] = @@TRANCOUNT --2 transações
    ,[XACT_STATE] = XACT_STATE()
      --Status 1 = Transação ativa
  END CATCH
 COMMIT TRAN
COMMIT TRAN
GO

SELECT * FROM #TEMP
DROP TABLE #TEMP
```

```
(1 row(s) affected)

@@TRANCOUNT XACT_STATE
----------- ----------
0           0


@@TRANCOUNT XACT_STATE
----------- ----------
2           1

(1 row(s) affected)

(1 row(s) affected)
(0 row(s) affected)

@@TRANCOUNT XACT_STATE
----------- ----------
2           1

X
-----------
1
2 -- ERA PARA ESTAR AQUI?
3 -- ERA PARA ESTAR AQUI?

(3 row(s) affected)
```

```
SET XACT_ABORT ON; -- NON DEFAULT

CREATE TABLE #TEMP (X INT)
INSERT INTO #TEMP VALUES (1)

SELECT
   [@@TRANCOUNT] = @@TRANCOUNT --0 transações
  ,[XACT_STATE] = XACT_STATE()
      --Status 0 = Nenhuma transação ativa

BEGIN TRAN
 BEGIN TRAN
  SELECT
    [@@TRANCOUNT] = @@TRANCOUNT --2 transações
   ,[XACT_STATE] = XACT_STATE()
      --Status 1 = Transação ativa

  INSERT INTO #TEMP VALUES (2) -- OK
  BEGIN TRY
   INSERT INTO #TEMP VALUES (3) -- OK
   INSERT INTO #TEMP VALUES (1/0) -- ERROR
  END TRY
  BEGIN CATCH
   SELECT
     [@@TRANCOUNT] = @@TRANCOUNT --2 transações
    ,[XACT_STATE] = XACT_STATE()
      --Status -1 = Transação NÃO pode ser
comitada
  END CATCH

 COMMIT TRAN
COMMIT TRAN
GO




SELECT * FROM #TEMP
DROP TABLE #TEMP
```

```
(1 row(s) affected)


@@TRANCOUNT XACT_STATE
----------- ----------
0           0


@@TRANCOUNT XACT_STATE
----------- ----------
2           1

(1 row(s) affected)

(1 row(s) affected)
(0 row(s) affected)


@@TRANCOUNT XACT_STATE
----------- ----------
2           -1


Msg 3930, Level 16, State 1,
Line 29
The current transaction cannot
be committed and cannot
support operations that write
to the log file. Roll back the
transaction.


X
-----------
1

(1 row(s) affected)
```

3.3 Nesting transactions

As transações nesting leva em consideração o primeiro comando de commit ou rollback de uma transação.

No exemplo abaixo é possível verificar o erro informando que não tem um BEGIN TRANSACTION, para o comando commit, por mais que tivéssemos dois comandos de BEGIN TRANSACTION, como o ROLLBACK está antes o comando commit não será executado.

O ROLLBACK neste caso será executado para todas as transações:

```
CREATE TABLE #TESTE (x int);

BEGIN TRANSACTION

       INSERT INTO #TESTE VALUES (4);

       BEGIN TRANSACTION

              INSERT INTO #TESTE VALUES
(5);

       ROLLBACK TRAN;
COMMIT TRANSACTION; --ERROR
GO


SELECT * FROM #TESTE
DROP TABLE #TESTE
```

```
(1 row(s) affected)

(1 row(s) affected)

Msg 3902, Level 16, State 1, Line 12
The COMMIT TRANSACTION request has no
corresponding BEGIN TRANSACTION.

x
-----------

(0 row(s) affected)
```

Caso comentássemos o comando ROLLBACK TRAN, este seria o resultado:

```
(1 row(s) affected)

(1 row(s) affected)
x
-----------
4
5

(2 row(s) affected)
```

Para evitar fazer o ROLLBACK para todas as transações, é possível criar pontos de salvamento nas transações. É possível criar pontos de salvamento com o mesmo nome, mas no momento do ROLLBACK, ele é feito apenas para o ponto mais recente.

No exemplo abaixo, é possível verificar que temos o ponto de salvamento da transação chamado de ProcedureSave, logo depois de inserir o valor 4, depois é inserido o valor 5 e feito um ROLLBACK da transação, e logo após é inserido o valor 6 e feito o COMMIT.

Dessa maneira, quando é feito o SELECT da tabela temporária os valores 4 e 6 são retornados, deixando de fora apenas o valor 5, no qual foi feito o ROLLBACK após o ponto de salvamento.

```
CREATE TABLE #TESTE (x int);

BEGIN TRANSACTION ProcedureSave

       INSERT INTO #TESTE VALUES (4);

       SAVE TRANSACTION ProcedureSave;

       INSERT INTO #TESTE VALUES (5);

       ROLLBACK TRAN ProcedureSave;

       INSERT INTO #TESTE VALUES (6);
COMMIT TRAN ProcedureSave



GO
SELECT * FROM #TESTE
DROP TABLE #TESTE
```

```
(1 row(s) affected)


(1 row(s) affected)


(1 row(s) affected)


x
-----------
4
6 --ONLY VALUE (5) is gone

(2 row(s) affected)
```

É possível utilizar todos os comandos em uma transação, exceto os abaixo:

ALTER DATABASE	LOAD DATABASE
BACKUP LOG	LOAD TRANSACTION
CREATE DATABASE	RECONFIGURE
DISK INIT	RESTORE DATABASE
DROP DATABASE	RESTORE LOG
DUMP TRANSACTION	UPDATE STATISTICS

Também não é possível utilizar a opção sp_dboption para configurar opções de banco de dados ou qualquer procedure de sistema que modificam o banco de dados master nas transações definidas por usuário.

4 Objeto SqlConnection

O objeto SqlConnection representa uma sessão única a um servidor SQL Server e é necessário para que uma aplicação acesse e execute comandos no banco de dados.

Quando uma transação é iniciada no banco de dados ela fica associada a uma conexão, portanto, o uso correto do objeto de conexão pode ajudar a mitigar problemas de lock decorrentes de transações não finalizadas.

Para garantir que uma transação não fique aberta por algum erro não tratado é recomendado que toda conexão seja fechada após seu uso.

No exemplo abaixo, o bloco **using** foi utilizado para garantir o fechamento da conexão com a chamada do método **Dispose**. Caso exista alguma transação aberta no momento do fechamento da conexão, ela será automaticamente desfeita (rollback).

Exemplo:

```
using (SqlConnection connection = new SqlConnection(connectionString))
  {
        connection.Open();
        // Do work here; connection closed on following line.

  }
```

Através do uso do Connection Pooling (definido na string de conexão) é possível evitar o fechamento de fato da conexão entre a aplicação e o SQL Server, pois na chamada dos métodos Close/Dispose a conexão é devolvida para o connection pool.

Manter apenas um objeto SqlConnection e compartilhá-lo na execução de diversos comandos permite que, em caso de alguma situação não tratada, comandos distintos sejam aninhados sob uma transação existente e não seja possível a execução do "commit". A alternativa é encerrar a aplicação, o que resultará em "rollback".

5 TransactionScope

O *TransactionScope* é a classe responsável por implicitamente criar um bloco de código transacional. Esta classe prove um modelo de programação no qual as transações são automaticamente gerenciadas pela infraestrutura, sem exigir que o desenvolvedor tenha de interagir com objetos e componentes complexos de transação.

Este modelo não exige que os recursos que suportem transações sejam pré-identificados, pois qualquer recurso que provê suporte às transações, e que esteja dentro do bloco de código transacional, será identificado automaticamente.

5.1 Criando um bloco transacional

O escopo da transação é criado a partir de uma nova instância de um *TransactionScope*. Recomenda-se que a criação deste bloco seja feita dentro de um bloco *using*, a fim de garantir que todos os recursos alocados para a criação da transação sejam corretamente liberados ao término de sua execução.

O Quadro 1 apresenta um exemplo de criação de uma instância de *TransactionScope*.

```
var scopeOption = TransactionScopeOption.Required;

var trnOptns = new TransactionOptions() {
    IsolationLevel = IsolationLevel.ReadCommitted
};

using (var tran = new TransactionScope(scopeOption, trnOptns)) {

    // Commands...

    tran.Complete();
}
```

Quadro 1: Exemplo de criação de bloco transacional com TransactionScope.

A execução do método *Complete()* concretiza a transação, e confirma a persistência das operações envolvidas. Caso o bloco transacional chegue ao final sem a execução do método *Complete()*, então a transação é abortada e as alterações feitas pelas operação não são persistidas.

5.2 Funcionamento

Quando uma nova instância de *TransactionScope* é criada, o gerenciador de transações determina como a transação deve ser criada. Uma vez determinado, o escopo sempre utilizará a mesma transação durante todo o bloco.

O comportamento de criação das transações pode ser alterado via parâmetros, a Tabela 1 apresenta os possíveis tipos de transações que podem ser criados.

Tabela 1: Tipos de transações e seus impactos sobre o bloco transacional.

Tipo de Transação	Definição
Required	Indica que a transação é obrigatória para o escopo. Este parâmetro indica que, caso já exista uma transação ela será reaproveitada. Senão, uma nova transação é criada. Este é o comportamento padrão.
RequiresNew	Uma nova transação é criada para cada transação, mesmo que já exista uma em execução.
Suppress	A transação é suprimida na criação do escopo. Todas as operações no escopo são feitas sem um contexto de transação.

5.3 Bloqueios

O uso do construtor padrão da classe *TransactionScope* é propenso a bloqueios, quando criado com o propósito de gerenciar transações com bancos de dados SQL Server. Isso ocorre pois, neste construtor assume-se que o nível de isolamento padrão é *Serializable*. *Serializable* é um nível de isolamento mais elevado que o nível de isolamento padrão em bancos de dados SQL Server, o *Read Commited*. Por conta deste comportamento, a possibilidade de termos uma quantidade maior de bloqueios em cenários de concorrência, devido ao nível de isolamento mais restrito, é maior.

Por essa razão recomenda-se que em toda criação de uma nova instância de *TransactionScope* não seja utilizado o construtor padrão, e sim uma assinatura que permita especificar qual o nível de isolamento que se deseja utilizar para o bloco transacional que será executado.

O Quadro 1 apresenta um exemplo de como é possível criar uma nova instância de *TransactionScope* definindo via parâmetros o nível de isolamento *Read Commited* para toda a transação.

6 Melhores práticas

É importante manter as transações o mais curta possível. Quando uma transação é iniciada, o banco de dados tem que manter os recursos envolvidos nela bloqueados até que ela seja comitada ou aconteça o rollback, ou seja, até o fim da transação. Para proteger as propriedades ACID da transação.

Se os dados são modificados, as linhas que serão modificadas deverão ter um lock exclusivo para proteger de outras transações que podem estar lendo, os locks exclusivos, devem ser mantidos até o comit ou rollback.

Dependendo o nível de isolamento da transação, alguns comandos SELECT podem adquirir locks que protegerão os dados até a transação sofre um comit ou rollback.

Especialmente em sistemas com muitos usuários, as transações devem ser o mais curta possível, para reduzir as contenções de lock entre recursos de transações concorrentes.

Queries com tempo de execução muito alto, podem não ser um problema em um ambiente com poucos usuários, mas quando temos milhares de usuários, isso pode ser intolerável.

Orientações para escrita de códigos

Estas orientações são para escrever códigos mais eficientes:

- Não solicite entradas dos usuários durante a transação.

Tenha todas as entradas dos usuários antes da transação iniciar. Se entradas adicionais são necessárias durante a transação, faça o rollback da transação atual e reinicie a transação depois da entrada do usuário. Mesmo que o usuário responda imediatamente, a reações humanas é bem mais devagar que a de um computador.

Todos os recursos mantidos pelas transações podem ser mantidos por um longo tempo, o que pode ter potencial para causar problemas de bloqueio. Se os usuários não responderem, a transação se mantém ativa e fazendo lock em recursos críticos, até que o usuário responda, o que pode levar vários minutos ou horas.

- Não abra a transação enquanto está procurando através dos dados, se for possível.

As transações não devem iniciar até que todas as análises preliminares sejam concluídas.

- Mantenha a transação o mais curta possível.

Depois de saber as modificações que devem ser feitas, abra a transação, execute os comandos de modificação, e então imediatamente faça o commit ou rollback da transação. Não abra a transação antes de requerida.

- Faça uso inteligente de nível de isolamentos mais baixos.

Muitas aplicações podem utilizar o nível de isolamento read-commited. Nem todas as transações exigem o nível de isolamento de transação serializable.

- Faça uso inteligente menores opções de concorrência, como as opções de concorrência otimista.

Em um Sistema com baixa probabilidade de updates concorrentes, o custo de lidar com erros do tipo "alguém alterou o dado depois de eu o ler", pode ser bem mais baixo do que o custo de linhas sofrendo locks enquanto estão sendo lidas.

- Acesse a menor quantidade de dados possíveis em uma transação.

Quanto menor o número de linhas com locks, menor é a contenção entre transações.

- Evite problemas de concorrência

Para evitar problemas de concorrência, gerencie as transações implícitas com cuidado. Quando usando transações implícitas, o comando Transact-SQL após o commit ou rollback, automaticamente inicia uma nova transação. Com isso uma nova transação pode ser aberta, enquanto a aplicação busca por dados, ou então quando requer entrada do usuário. Depois de terminada a última transação, é necessário para proteger as modificações de dados desativar as transações implícitas, até que uma transação precise novamente proteger dados. Este processo deixa o Microsoft SQL Server utilizar o modo autocommit enquanto a aplicação está procurando dados e a entrada do usuário.

Ao utilizar a classe TransactionScope, procura definir via seu construtor o nível de isolamento desejado. Além disso, inclua no bloco transacional apenas comandos que realmente sejam necessários para a transação, a fim de diminuir sua complexidade e seu tempo de vida.

7 Referências

Transactions (Database Engine)
https://technet.microsoft.com/en-us/library/ms190612(v=sql.105).aspx

SET IMPLICIT_TRANSACTIONS (Transact-SQL)
https://msdn.microsoft.com/en-us/library/ms187807.aspx

Transaction Statements (Transact-SQL)
https://msdn.microsoft.com/en-us/library/ms174377.aspx

SET XACT_ABORT (Transact-SQL)
https://msdn.microsoft.com/en-us/library/ms188792.aspx

XACT_STATE (Transact-SQL)
https://msdn.microsoft.com/en-us/library/ms189797.aspx

@@TRANCOUNT (Transact-SQL)
https://msdn.microsoft.com/en-us/library/ms187967.aspx

SqlConnection Class
https://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqlconnection(v=vs.110).aspx

SET TRANSACTION ISOLATION LEVEL (Transact-SQL)
https://msdn.microsoft.com/pt-br/library/ms173763(v=sql.120).aspx

IsolationLevel Property (ADO)
https://msdn.microsoft.com/en-us/library/ee252416(v=bts.10).aspx

OleDbTransaction.IsolationLevel Propriedade
https://msdn.microsoft.com/pt-br/library/system.data.oledb.oledbtransaction.isolationlevel(v=vs.110).aspx?cs-save-lang=1&cs-lang=vb#code-snippet-1

TransactionScopeOption Enumeration
https://msdn.microsoft.com/en-us/library/system.transactions.transactionscopeoption(v=vs.110).aspx

Implementing an Implicit Transaction using Transaction Scope
https://msdn.microsoft.com/en-us/library/ee818746(v=vs.110).aspx

TransactionScope.Complete Method()
https://msdn.microsoft.com/en-us/library/system.transactions.transactionscope.complete(v=vs.110).aspx