Desenvolvimento de Sistemas

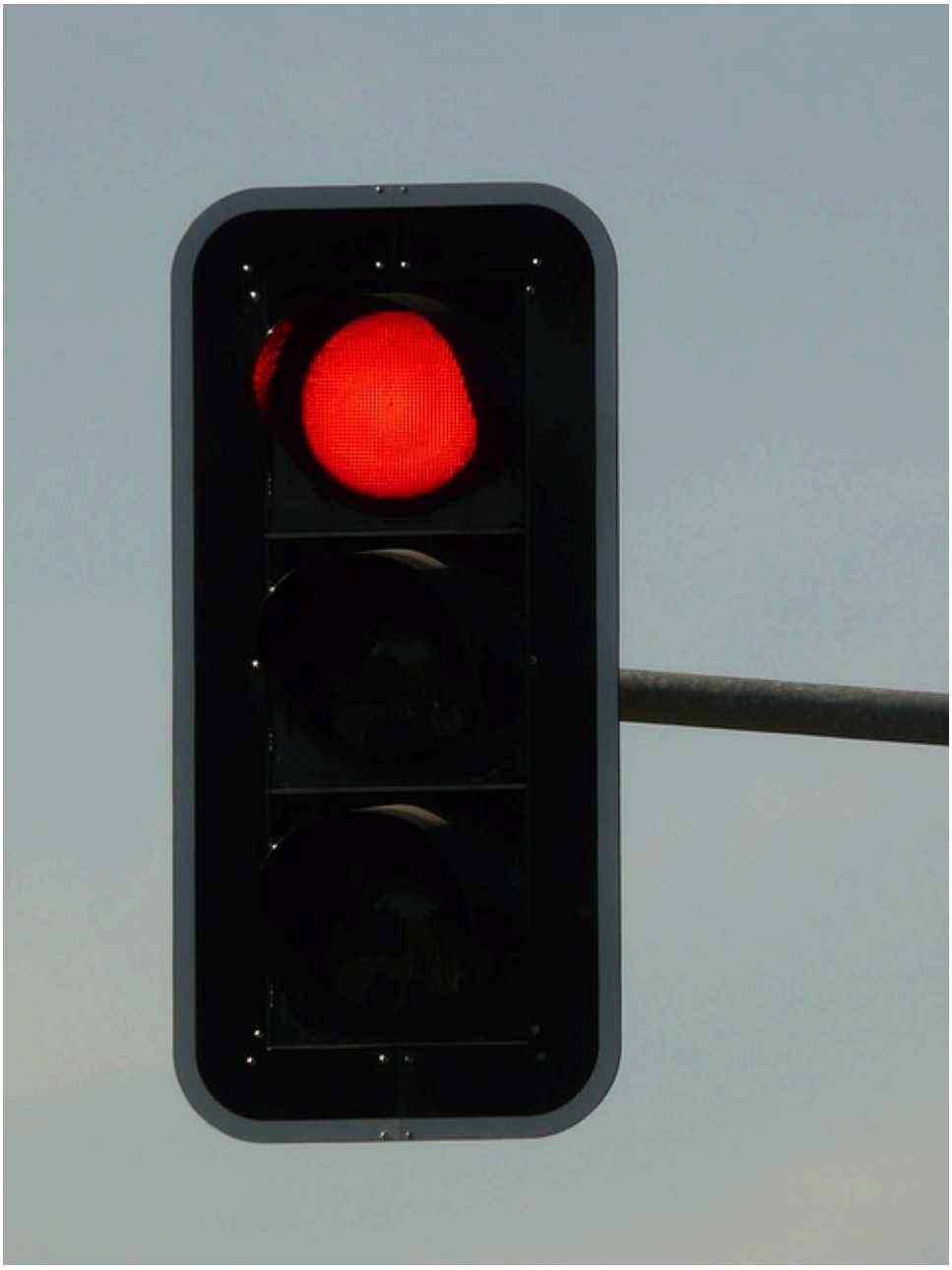
Transações e *locks*

Dois dos conceitos mais importantes e presentes, mesmo implicitamente, em quase todas as operações aplicadas a banco de dados são os de bloqueios e de transações. Os bloqueios (ou *locks*) auxiliam na consistência de dados, garantindo que a inclusão de um registro em uma tabela, por exemplo, ocorra exatamente como especificado pela instrução SQL (*structured query language*) executada e impedindo que outras operações paralelas afetem negativamente o resultado desse comando.

A seguir, aprenda sobre o que são bloqueios e transações e como manipulá-las manualmente no MySQL, além de quando aplicá-las e quando evitá-las.

Conceitos

Um banco de dados naturalmente é desenvolvido de maneira que múltiplas operações simultâneas sejam realizadas nele. Imagine o quão inadequado seria um banco que permitisse apenas uma conexão por vez. Isso impossibilitaria a operação até de sistemas muito simples – basta haver mais de um usuário ao mesmo tempo.



1 – *Locks* atuam nos dados como semáforos de trânsito
Fonte: <https://pixabay.com/pt/photos/sem%c3%a1foro-luzes-de-tr%c3%a2nsito-vermelho-8511/>. Acesso em: 23 mar. 2022.

A figura mostra um semáforo com luz vermelha acesa.

Essas operações simultâneas, ou concorrências, no entanto, trazem uma série de consequências, sendo a mais imediata o risco de que uma operação afete negativamente a outra – é o caso em que dois usuários estão executando

inserções e exclusões em uma mesma tabela. Sem um controle, o cenário pode se tornar caótico, misturando ou perdendo dados, uma vez que são várias instâncias tentando escrever um mesmo arquivo.

Neste contexto, um dos elementos básicos para controle de concorrência é o bloqueio, ou ***lock***. Trata-se de um mecanismo dos sistemas de banco de dados que “tranca” momentaneamente enquanto o banco é manipulado. Uma analogia comum que se faz é com o trânsito: assim como um semáforo que regula os carros em uma interseção evitando que os veículos cruzem uns nos caminhos dos outros e colidam, os *locks* garantem que o acesso a dados de duas consultas simultâneas não conflite.

Existem diferentes níveis de bloqueios, sendo que, por padrão, no mecanismo InnoDB do MySQL implemente-se o **bloqueio de registro**, que protege apenas o registro que está sendo manipulado por uma operação SQL, deixando o restante dos registros da mesma tabela livre. No entanto, há bloqueios que protegem toda a tabela e até mesmo bloqueios ao nível de usuário e de estrutura – que impedem alterações simultâneas no formato de uma tabela, por exemplo. *Locks* específicos também podem ser definidos pelo usuário para tabelas específicas, o que fornece um controle maior da concorrência observada em um sistema, permitindo ajustar situações problemáticas de *performance* ou de inconsistência de dados em um banco.

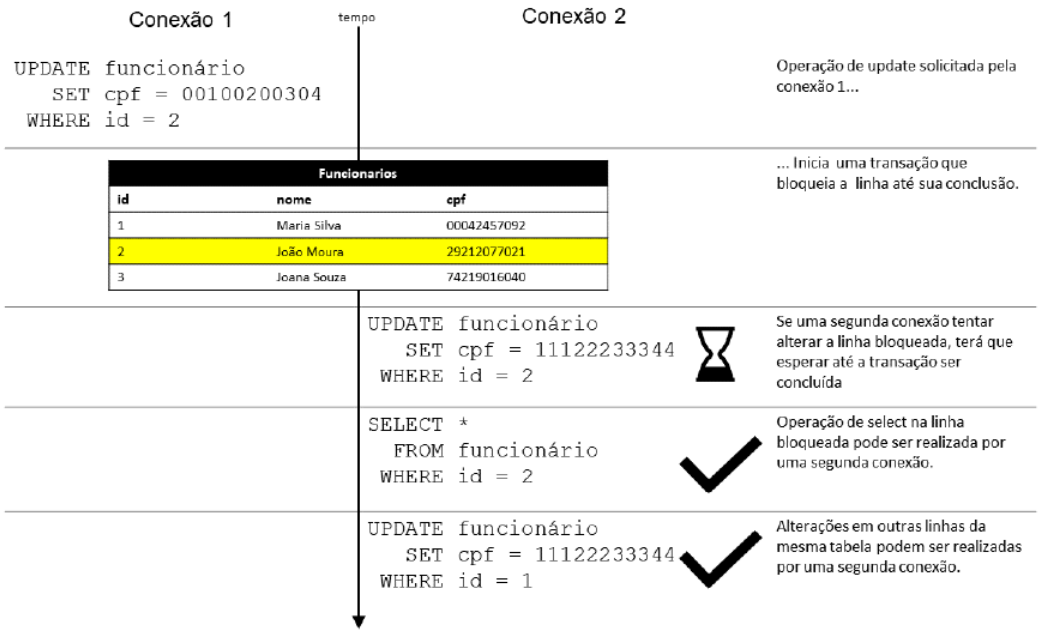


Figura 2 – Funcionamento do *row-level lock*, ou bloqueio de registro, padrão da arquitetura InnoDB no MySQL

A figura mostra uma linha do tempo ao centro apontando para baixo. À sua esquerda estão a "conexão" e o *script* "UPDATE funcionário SET cpf = 00100200304 WHERE id = 2"; à direita está a explicação "operação de update solicitada pela conexão 1...". Abaixo, há uma tabela "funcionários" com três registros, mostrando que o registro de id = 2 está bloqueado; à direita consta a explicação "... Inicia uma transação que bloqueia a linha até sua conclusão.". Abaixo, à direita da linha do tempo, está o *script* "UPDATE funcionário SET cpf = 11122233344 WHERE id = 2" e, à direita, a explicação "Se uma segunda conexão tentar alterar a linha bloqueada, terá que esperar até a transação ser concluída". Abaixo, à direita da linha do tempo, está o SQL "SELECT * FROM funcionário WHERE id = 2" e a explicação "Operação de select na linha bloqueada pode ser realizada por uma segunda conexão.". Abaixo, à direita da linha do tempo, está o SQL "UPDATE funcionário SET cpf = 1122233344 WHERE id = 1" e a explicação "Alterações em outras linhas da mesma tabela podem ser realizadas por uma segunda conexão".

Um conceito muito semelhante ao de bloqueios é o das **transações** (ou *transactions*). Trata-se do mecanismo que permite que um conjunto de operações de banco de dados se comporte atomicamente, ou seja, como se fosse uma única operação. As consequências dessas operações realizadas (as modificações em dados, por exemplo) em uma transação podem ser aplicadas ou rejeitadas. As transações são importantes para manter a integridade de dados, mas, quando utilizadas incorreta ou inconsequentemente, podem causar lentidão ou travar recursos do banco de dados.

As transações contêm quatro propriedades padrão, apelidadas de ACID (atomicidade, consistência, isolamento e durabilidade). Veja agora cada uma delas:

Atomicidade

Assegura que todas as operações em uma transação sejam concluídas com sucesso. Caso contrário, a transação é abortada e as operações realizadas desde o início da transação são desfeitas.

Consistência

Assegura que o banco de dados seja alterado apropriadamente após a conclusão bem-sucedida da transação.

Isolamento

Cada transação é isolada de outra, ou seja, são executadas independente e transparentemente.

Durabilidade

Garante que o resultado ou efeito de uma transação bem-sucedida persista no banco de dados e não seja perdido.

As transações são acompanhadas de duas operações que as concluem:

COMMIT

Finalização de uma transação bem-sucedida. Realiza a persistência em banco de dados do resultado das operações realizadas durante a transação. Imagine que, em uma transação, tenha-se solicitado a inclusão de dados e a atualização de alguns registros de uma tabela. Ao realizar **COMMIT**, esses dados serão efetivamente incluídos e atualizados.

ROLLBACK

Finalização de uma transação malsucedida. Desfaz as operações de uma transação, retornando o banco de dados ao estado imediatamente anterior ao início das transações. Imagine que, em uma transação, realize-se a inclusão de um registro em uma tabela e a exclusão de outro. Ao finalizá-la com **ROLLBACK**, os dados incluídos não persistirão na tabela e a exclusão não ocorrerá, fazendo com que a tabela afetada volte a apresentar a linha excluída.

Na prática, quando se executa uma operação SQL, o sistema de banco de dados está criando pequenas transações e finalizando-as assim que concluídas as operações. Nessas transações é que acontecem os bloqueios.

Obviamente, podem-se definir bloqueios e transações manualmente. No entanto, apesar da grande utilidade das transações, é necessário cuidado na hora de utilizá-las. Sabe-se que, durante uma transação, várias operações SQL podem ser realizadas; quanto mais operações, por conta da propriedade de atomicidade das transações, mais recursos são requeridos e bloqueados. Quanto mais tempo a transação durar, maiores serão os *locks*. Além disso, registros de *logs* para desfazer as operações em caso de **ROLLBACK** serão mais numerosos. Com esses recursos presos, outras consultas paralelas podem simplesmente travar à espera da liberação, o que pode resultar em situações de *timeout* (tempo de espera esgotado) ou *deadlock* (o que será discutido adiante). Assim, é necessário cuidado com o planejamento e a finalização de uma transação.

Comandos e aplicações

Utilizando transações

De modo geral, o uso de transações ocorre com os seguintes passos:

1. Iniciar a transação
2. Executar um ou mais comandos SQL, como **SELECT**, **INSERT**, **UPDATE** ou **DELETE**, ou chamadas as funções e os procedimentos
3. Checar se há algum erro ou inconsistência no resultado
4. Se houver erro, realizar **ROLLBACK**; caso contrário, acionar **COMMIT**

No MySQL e na maioria dos SGBDs (Sistemas de Gerenciamento de Banco de Dados), a sintaxe para transações é a seguinte:

1. **START TRANSACTION** ou **BEGIN** para iniciar uma transação
2. **COMMIT** para encerrar com sucesso, persistindo os dados
3. **ROLLBACK** para encerrar com falha, retornando o banco ao estado anterior

Para executar os exemplos a seguir, será utilizado como base um banco de dados muito simples, com uma tabela de vendedores e outra de vendas.

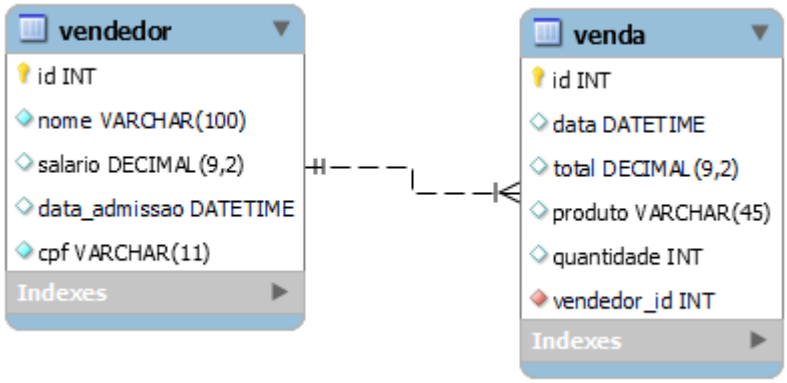


Figura 3 – Modelo ER do banco de dados para testes

A figura mostra uma caixa para a tabela “vendedor”, à direita, com colunas “id int”, que é chave, “nome varchar(100)”, “salario decimal(9,2)”, “data_admissao datetime” e “cpf varchar(11)”. À direita, ligada por uma relação 1:N com a tabela vendedor, está a tabela “venda”, com colunas “id int”, que é chave, “data datetime”, “total decimal(9,2)”, “quantidade int” e “vendedor_id int”, que é referência.

Crie, então, no MySQL, um banco de dados chamado “teste_locks” e execute o *script* a seguir com esse novo banco ativado:

```
CREATE TABLE vendedor (
    id INT NOT NULL AUTO_INCREMENT,
    nome VARCHAR(100) NOT NULL,
    salario DECIMAL(9,2) NULL,
    data_admissao DATETIME NULL,
    cpf VARCHAR(11) NOT NULL,
    PRIMARY KEY (`id`))
ENGINE = InnoDB;

CREATE TABLE venda (
    id INT NOT NULL AUTO_INCREMENT,
    data DATETIME NULL,
    total DECIMAL(9,2) NULL,
    produto VARCHAR(45) NULL,
    quantidade INT NULL,
    vendedor_id INT NOT NULL,
    PRIMARY KEY (id),
        FOREIGN KEY (vendedor_id)
        REFERENCES vendedor (id))
ENGINE = InnoDB;
```

Com o banco criado e selecionado, usando o MySQL Workbench, crie uma nova aba SQL (menu **File** > **New Query Tab**) e aplique o seguinte *script*:

```
START TRANSACTION;

INSERT INTO vendedor (nome, salario, data_admissao, cpf)
VALUES ('José da Silva', 1500, '2020-10-10', '00100200304');

SELECT * FROM vendedor;

ROLLBACK;

SELECT * FROM vendedor;
```

Execute esse *script* por meio do ícone de raio (ou atalho **Ctrl** + **Shift** + **Enter**) sem ter selecionado nenhuma linha no editor, para que o *script* seja executado do início ao fim.

Nesse *script*, você estará:

- Iniciando uma transação com **START TRANSACTION**. A partir do momento em que essa instrução é executada, as operações SQL que forem executadas nessa conexão estarão protegidas pela transação.
- Incluindo um registro na tabela “vendedor” com uma instrução **INSERT**.
- Verificando os dados da tabela “vendedor” com a instrução **SELECT**. Note que essa instrução mostrará os dados incluídos na instrução anterior.
- Encerrando a transação com **ROLLBACK**. Como você deve saber, essa instrução desfaz todas as alterações realizadas durante a transação. Assim, é esperado que o **INSERT** executado seja revertido.
- Verificando novamente os dados de “vendedor” com a instrução **SELECT**. Essa instrução deve mostrar que a tabela está vazia, por conta do **ROLLBACK**.

Evidentemente, é possível executar linha a linha do *script* de exemplo. Ou seja, a partir do momento em que **START TRANSACTION** for executada, a transação está aberta e qualquer SQL executado no editor estará dentro da transação.

Como resultado, o Workbench mostrará na seção **Result Grid** duas abas. A primeira corresponde ao primeiro **SELECT** e a segunda correspondente ao último **SELECT**.

A figura a seguir apresenta, à esquerda, o resultado da consulta executada dentro da transação e, à direita, o resultado após o **ROLLBACK**.

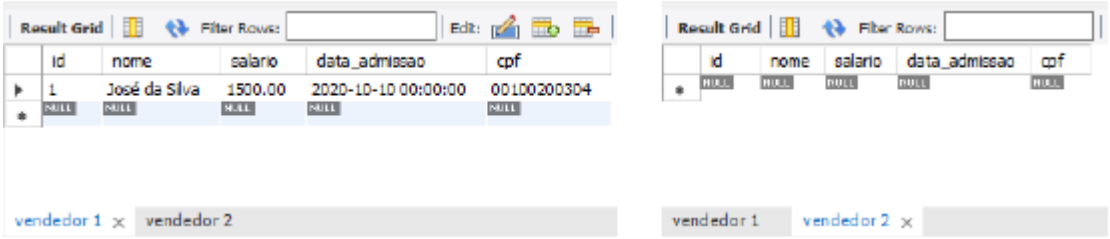


Figura 4 – Resultados para as duas instruções **SELECT** executadas

A figura mostra, à esquerda, uma linha com os nomes das colunas "id", "nome", "salario", "data_admissao" e "cpf" e, abaixo dessa linha, outra, com os dados "1", "José da Silva", "1500.00", "2020-10-10 00:00:00", "00100200304". Abaixo disso, consta uma linha com "null" em todas as colunas. À direita, está a tabela com nomes das colunas e com apenas uma linha com "null" em todas as colunas.

Como se pode notar na figura 4, os dados são incluídos na tabela, mas apenas durante a transação. Ao executar **ROLLBACK**, a tabela “vendedor” volta a seu estado original, sem dados.

Caso você troque a finalização por **COMMIT**, verá que a tabela persistirá com as modificações:

```
START TRANSACTION;

INSERT INTO vendedor (nome, salario, data_admissao, cpf)
VALUES ('José da Silva', 1500, '2020-10-10', '00100200304');

SELECT * FROM vendedor;

COMMIT;

SELECT * FROM vendedor;
```

Há praticamente o mesmo *script*, apenas trocando **ROLLBACK** por **COMMIT**. O resultado, neste caso, como esperado e como mostrado na figura 5, é de tabela com dados tanto no **SELECT** interno à transação quanto no externo. Observe que, à esquerda, está o resultado da consulta executada dentro da transação e, à direita, está o resultado após o **COMMIT**. Ambas trazem dados.

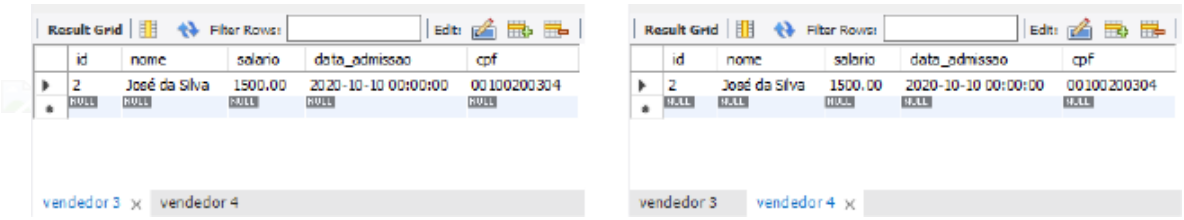


Figura 5 – Resultados para as duas instruções **SELECT** executadas

A figura mostra, à esquerda, uma linha com os nomes das colunas "id", "nome", "salario", "data_admissao", "cpf" e, abaixo, uma linha com os dados "2", "José da Silva", "1500.00", "2020-10-10 00:00:00", "00100200304". Abaixo disso, consta uma linha com "null" em todas as colunas. À direita, está uma tabela com os nomes das colunas e os mesmos dados mostrados à esquerda.

Não importa a quantidade de comandos SQL executados após o início da transação, eles serão todos persistidos ou anulados de acordo com a finalização utilizada.

Autoincremento foge às regras de **ROLLBACK**. Note, pelo exemplo, que a **id** de vendedor persistida não é 1, apesar de ser o primeiro registro na tabela. Isso porque as inserções executadas anteriormente foram revertidas.

No exemplo a seguir, inclui-se uma cláusula **INSERT**, que gerará erro de referência

```
START TRANSACTION;

INSERT INTO vendedor (nome, salario, data_admissao, cpf)
VALUES ('Maria Quintino', 1500, '2021-01-10', '00200300405');

SELECT * FROM vendedor;

INSERT INTO venda (data, total, produto, quantidade, vendedor_id)
VALUES ('2021-01-11', 100.5, 'kit mouse e teclado', 1, 100);

COMMIT;
```

Veja que, quando se faz inserção na tabela de vendas, utiliza-se uma **id** de vendedor que ainda não existe (“vendedor_id = 100”), o que acaba gerando uma falha de referência de chave estrangeira. Nesse caso, finalize com **COMMIT** e, assim, o registro da vendedora “Maria Quintino” será persistido normalmente. Isto é, o erro de SQL não anula a transação automaticamente, é preciso executar **ROLLBACK** manualmente, quando necessário.

Uma situação comum, principalmente na execução de testes, é a abertura de uma transação e a execução, uma por uma, de cada operação SQL, executando **COMMIT** somente com a certeza de que tudo está em ordem. **ROLLBACK**, neste caso, é um grande aliado, permitindo a reversão das operações para seguir com mais testes.

Transações bloqueiam os recursos afetados. Para testar, são necessárias duas conexões ao mesmo banco. Para isso, volte à página inicial (ícone de casa no canto superior esquerdo) e clique duas vezes na conexão local.

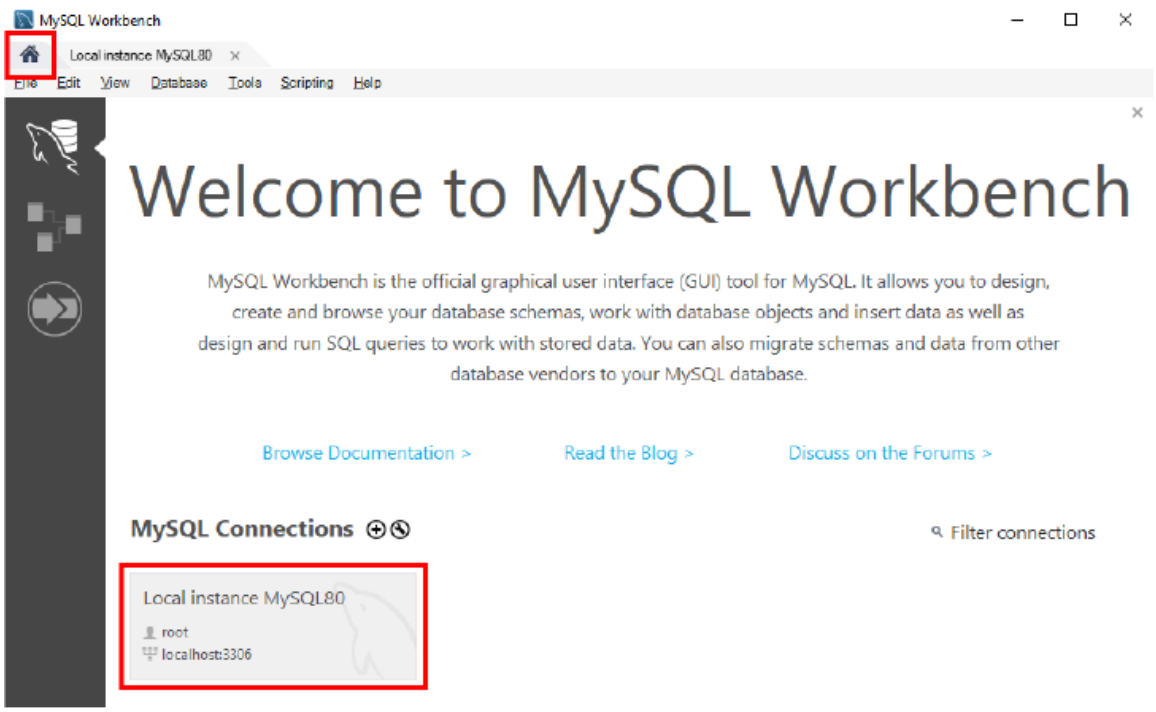


Figura 6 – Realizando uma segunda conexão

Tela inicial do Workbench, com destaque para o ícone superior esquerdo de casa e para o botão de conexão local, abaixo de “MySQL Connections”. Na tela exemplo, há apenas um botão de conexão com o título “Local instance MySQL80”.

Essa nova conexão deve abrir uma nova aba na parte superior do Workbench.

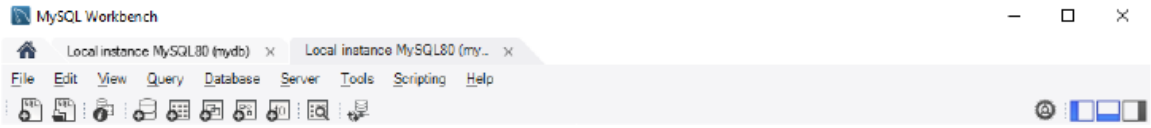


Figura 7 – Duas abas de conexão no Workbench

Topo da tela principal do Workbench com duas abas, cujo título é “Local Instance MySQL80”.

Agora que há duas conexões, selecione a primeira aba (conexão 1) e utilize o seguinte *script*.

```
START TRANSACTION;

UPDATE vendedor
    SET salario = 1600
 WHERE nome = 'José da Silva';

SELECT * FROM vendedor;
```

Note que não há finalização na transação e isso é proposital. Na segunda aba (conexão 2), acione o banco “teste_locks” clicando duas vezes sobre ele na área **Schemas**, à esquerda, ou usando o comando **USE teste_locks**. Em seguida, execute o *script* a seguir:

```
START TRANSACTION;

UPDATE vendedor
    SET salario = 1400, cpf='11122233344'
 WHERE nome = 'José da Silva';

SELECT * FROM vendedor;
```

Agora, existe um impasse. Note que, na segunda conexão, a operação não se conclui, como se pode observar na área **Output** e na aba **Query**.

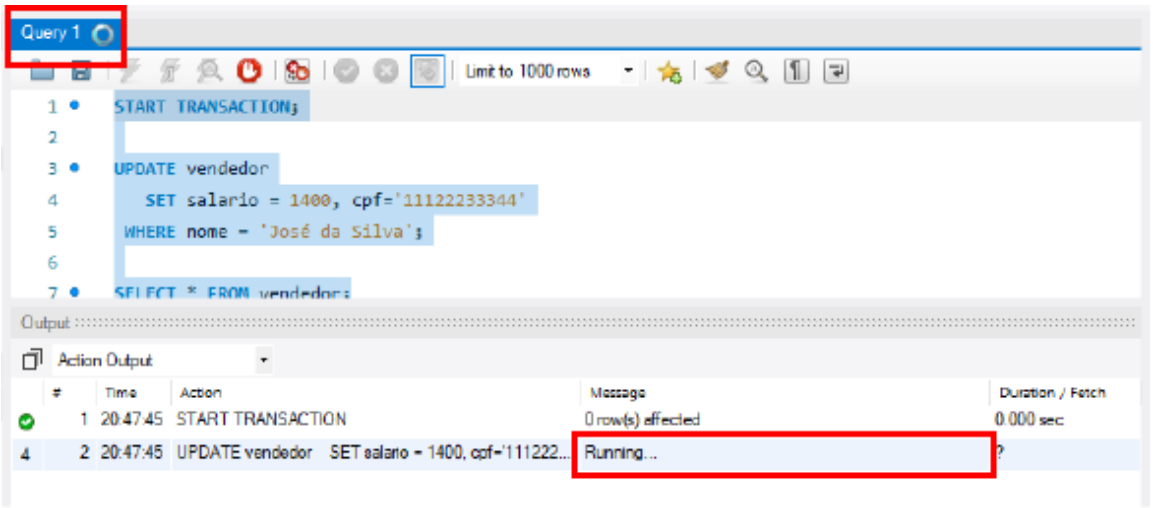


Figura 8 – Operação que não se conclui na segunda conexão

Tela principal do Workbench com uma aba “Query 1” e um ícone de espera à direita do título da aba. Abaixo, consta o *script* do exemplo. Abaixo, na área “Output”, a última linha mostra na coluna “Message” com o texto “Running...”.

Após alguns segundos (cerca de 30, pela configuração padrão), a conexão será derrubada e o Workbench solicitará reconexão. Em **Output** poderá surgir a mensagem “Error Code: 2013. Lost connection to MySQL server during query”.

Observa-se aqui o *lock* em ação. Note que, na primeira conexão, está sendo “trancado” o registro do funcionário “José da Silva” na tabela “vendedor” por conta da operação de **UPDATE** solicitada, na qual há a intenção de atualizar

o salário para 1600. Na segunda conexão, tem-se a intenção de atualizar o mesmo registro de “José da Silva” na mesma tabela “vendedor”. Como a transação na primeira conexão não foi finalizada, a segunda conexão não é capaz de concluir a operação – e, por conta da configuração de *timeout* (tempo de espera), a conexão é interrompida.

Esse é um fator de grande importância e que exige muita atenção ao usar transações: você está trancando recursos que outras conexões podem necessitar e, com isso, pode causar travamentos, lentidões ou falhas. Por isso, tenha muito cuidado para não deixar transações sem conclusão.

No exemplo, você voltará à conexão 1 e executará apenas a seguinte operação:

```
ROLLBACK;
```

Depois disso, execute normalmente o *script* presente na aba 2 e finalize-o com uma instrução **COMMIT**. Os dados de salário e CPF do vendedor são então persistidos no banco de dados.

Por padrão, os bloqueios no sistema InnoDB do MySQL trancam apenas a linha afetada na tabela. Para testar, execute o seguinte *script* na conexão 1:

```
START TRANSACTION;

UPDATE vendedor
    SET salario = 1600
 WHERE id = 2;

SELECT * FROM vendedor;
```

Na conexão 2, execute o seguinte *script*:

```
START TRANSACTION;

UPDATE vendedor
    SET salario = 1700
 WHERE id = 4;

SELECT * FROM vendedor;
```

A diferença deste com relação aos exemplos anteriores é que aqui você está utilizando a **id** na seleção do **UPDATE**. Note que ambas as operações são executadas com sucesso, já que uma não interfere na outra – a conexão 2 não necessita do recurso bloqueado (a linha de chave primária 2) da conexão 2, mas sim de outra linha (com chave primária 4).

O exemplo considera as **ids** 2 e 4 por conta da configuração da tabela na máquina do autor no momento dos testes:

#id	nome	salario	data_admissao	cpf
2	José da Silva	1600.00	10/10/2020 00:00	11122233344
4	Maria Quintino	1500.00	10/01/2021 00:00	00200300405

Se necessário, modifique os *scripts* para utilizar os valores de **id** presentes em seu banco de dados.

Se você mantiver o **UPDATE** selecionando dados pelo nome do vendedor, ainda assim terá bloqueio. Isso porque se trata de uma coluna que não é chave primária e, portanto, poderia ser modificada por qualquer uma das conexões.

Locks manuais

Além das transações, é possível realizar o bloqueio explícito de uma tabela por meio do comando **LOCK TABLES**.

LOCK TABLES tabela [READ | WRITE];

Para exemplificar, bloqueie a tabela “vendedor” na conexão 1 usando o seguinte *script*:

```
LOCK TABLES vendedor READ;
```

A palavra **READ** indica que a tabela está bloqueada para apenas leitura. A conexão poderá realizar consultas, mas não alterações de dados. As outras conexões ficam com alterações de dados bloqueadas.

Assim, a seguinte consulta resultará em sucesso:

```
SELECT * FROM vendedor;
```

Observe agora esta instrução:

```
INSERT INTO vendedor (nome, salario, data_admissao, cpf) VALUES ('Joaquim S
á', '2000', '2019-05-10', '00300400506');
```

Caso você execute essa instrução, obterá a mensagem de erro “Error Code: 1099. Table 'vendedor' was locked with a READ lock and can't be updated”, que significa que “a tabela vendedor estava bloqueada com READ e não pode ser atualizada”.

Já na conexão 2, se você executar o *script* de **INSERT** citado, o editor ficará em modo de espera.

Para liberar a tabela, utilize o seguinte comando:

```
UNLOCK TABLES;
```

Há, também, ainda outro tipo de bloqueio: o de escrita. Para testá-lo, utilize na primeira conexão a seguinte instrução:

```
LOCK TABLES vendedor WRITE;
```

Nesse tipo de bloqueio, a conexão poderá realizar leitura e escrita de dados na tabela. As outras conexões ficam bloqueadas.

Execute então as seguintes instruções SQL na conexão 1.

```
SELECT * FROM vendedor;
INSERT INTO vendedor (nome, salario, data_admissao, cpf) VALUES ('Antônio Mar
tins', '1850', '2020-01-05', '00400500607');
```

Note que ambas as instruções são bem-sucedidas.

Na conexão 2, mesmo que você execute o comando **SELECT** referido antes, a operação ficará bloqueada, com mensagem a “Running” até que a tabela seja desbloqueada na primeira conexão ou até que ocorra *timeout*.

Novamente, para desbloquear a tabela, utilize **UNLOCK TABLES**.

É importante ressaltar que bloqueios manuais não correspondem a transações, uma vez que em *locks* não se aplica uma finalização **ROLLBACK** para desfazer operações. De qualquer maneira, é possível usar **LOCK TABLE** juntamente com transações.

Na sessão com **LOCK**, a manipulação de tabela será restringida às tabelas explicitamente bloqueadas. Ao tentar realizar alteração de dado em outra tabela, será emitida a mensagem “Error Code: 1100. Table <name> was not locked with LOCK TABLES”, que significa que a “tabela ‘nome’ não foi bloqueada com LOCK TABLES”). É possível, nesses casos, realizar o desbloqueio ou o bloqueio de várias tabelas com a seguinte sintaxe:

LOCK TABLES tabela1 [READ | WRITE], tabela2 [READ | WRITE]...

Ao bloquear uma tabela manualmente com **LOCK TABLES WRITE**, seus relacionamentos (referências por chave estrangeira) também serão bloqueados.

Deadlock

Em algumas situações, os bloqueios acabam se tornando cíclicos, ou seja, uma cadeia de recursos que dependem uns dos outros, de maneira que o último recurso acaba dependendo do primeiro. Por exemplo, imagine que uma transação bloqueie uma tabela A e dependa da tabela C, uma segunda transação bloqueie a tabela B e dependa da tabela A e uma terceira transação bloqueie a tabela C e dependa da tabela B.

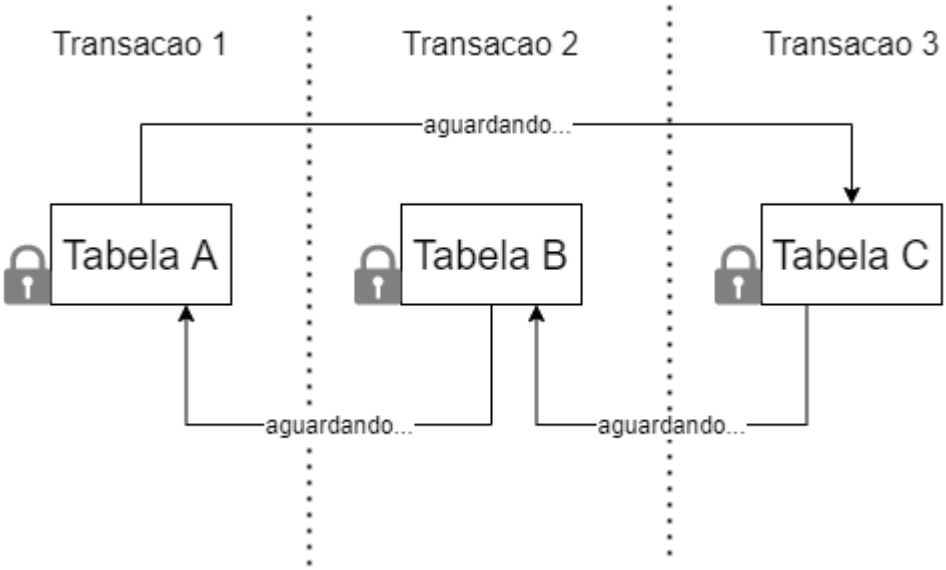


Figura 9 – Situação de *deadlock*, na qual as dependências se tornam cíclicas

A figura mostra três seções verticais, lado a lado: “Transação 1”, “Transação 2”, “Transação 3”. A Transação 1 tem um retângulo nomeado de “Tabela A”, com um cadeado, e desse retângulo parte uma flecha para a “Tabela C”, com a legenda “aguardando”. A Transação 2 tem um retângulo nomeado de “Tabela B”, com um cadeado, e desse retângulo parte uma flecha para a “Tabela A”, com a legenda “aguardando”. A Transação 3 tem um retângulo nomeado de “Tabela C”, com um cadeado, e desse retângulo parte uma flecha para a “Tabela B”, com a legenda “aguardando”. Forma-se assim um ciclo de dependência entre as tabelas.

Esse tipo de cenário é chamado de ***deadlock***, uma situação de bloqueio insolúvel, a menos que uma das conexões desista de seu bloqueio. *Deadlocks* são como cruzamentos congestionados, em que nenhum veículo é capaz de avançar (ao menos que alguns carros resolvam mudar seus trajetos).

Figura 10 – Cruzamento congestionado

Fonte: <https://cs.nyu.edu/~gottlieb/courses/2000s/2007-08-spring/os/lectures/diagrams/gridlock.jpg>. Acesso em: 23 mar. 2022.

A figura mostra um cruzamento de ruas com muitos carros em todas as direções, de maneira que um eles trancam a passagem uns dos outros.

Para entender melhor, considere criar deliberadamente uma situação de dependência para gerar *deadlock*. Primeiro, em uma conexão, inclua um novo registro de vendedor e de venda.

```
START TRANSACTION;

INSERT INTO vendedor (nome, salario, data_admissao, cpf)
VALUES ('Jerônimo Sá', 1250.75, '2020-01-10', '00500600708');

/*encontrando o último id de vendedor e armazenando em uma variável*/
SELECT @id_vendedor := MAX(id) FROM vendedor;

INSERT INTO venda (data, total, produto, quantidade, vendedor_id)
VALUES ('2021-01-11', 100.5, 'kit mouse e teclado', 1, @id_vendedor);

COMMIT;
```

Serão necessárias as duas conexões estabelecidas nos exemplos anteriores.

Conexão 1	Conexão 2
Na primeira conexão, inicie uma transação e atualize o registro do novo vendedor. O **UPDATE** bloqueará o registro afetado.	

`START TRANSACTION;`
 	
`UPDATE vendedor SET salari o = 1300 WHERE id = 9;`
`/*use o valor de id de "Je rônimo Sá" em seu banco de dados*/`	

Na segunda conexão, inicie uma transação e exclua a nova venda. O **DELETE** bloqueará o registro afetado.	

	`START TRANSACTION;`
	
	`DELETE FROM venda WHERE id = 2;`
	`/*use o valor de id da últ ima venda incluída em seu banco de dados*/`

Retornando à primeira conexão, atualize a venda incluída por último. No **Output**, aparecerá a mensagem “Running...”, indicando o estado de espera.	

`UPDATE venda`
`SET produto = 'teclado + m ouse' WHERE id = 2;`
`/*use o valor de id da últ ima venda incluída em seu banco de dados*/`	

Rapidamente, na segunda conexão, execute um **UPDATE** no registro de “Jerônimo Sá”, na tabela de vendedor. Essa execução deve acontecer antes do *timeout* (30 segundos, como padrão)	

	`UPDATE vendedor`
	`SET cpf = '94357011086'`
	`WHERE id = 9;`
	`/*use o valor de id de "Jerônimo Sá" em seu banco de dados*/`

O *deadlock* ocorre, então execute **ROLLBACK** em ambas as conexões.	

Após executar essa série de passos e nessa ordem, você obterá no **Output** de uma das conexões (nesse exemplo, geralmente na conexão 1) a seguinte mensagem: “Error Code: 1213. Deadlock found when trying to get lock; try restarting transaction”, que significa “deadlock encontrado ao tentar obter bloqueio; tente reiniciar a transação”. Na outra conexão, a operação SQL foi bem-sucedida. Esta é uma maneira de o MySQL tratar *deadlock*: invalidar a operação causadora do *deadlock* em uma das conexões, abrindo caminho para a outra conexão.

Outro recurso que o SGBD implementa para evitar *deadlock* é o próprio *timeout*. Caso a espera por um bloqueio seja acima do tempo configurado, simplesmente interrompe-se a conexão. O bloqueio por registro, implementado pelo InnoDB, também auxilia a evitar *deadlocks*, já que o bloqueio de uma tabela inteira abriria mais espaço para *locks* cruzados.

Ao programar um banco de dados, especialmente ao implementar rotinas de *procedures* e *functions*, é importante investigar se tais situações de bloqueios cruzados podem acontecer.

O MySQL provê algumas ferramentas administrativas que podem auxiliar no diagnóstico e no tratamento de problemas com bloqueios e *deadlocks*. Algumas dessas ferramentas são os seguintes comandos:

SHOW PROCESSLIST

Lista as operações (ou processos) que estão ocorrendo neste momento nas conexões ao banco de dados. Pode ser útil para revelar estados de espera de uma conexão com outra (experimente utilizar em uma situação de bloqueios de tabelas, obtendo a mensagem “Waiting for table metadata lock” em um dos processos).

SHOW ENGINE INNODB STATUS

Relatório de *status* do mecanismo de banco de dados, detalhando transações e operações ocorridas.

KILL <número do processo>

Utilizado em conjunto com os itens anteriores, permite encerrar um processo, interrompendo um bloqueio.

Que tal realizar alguns desafios?

Agora que você já conhece as ferramentas e os elementos de segurança que permitem concorrência em operações de banco de dados, escreva *scripts* SQL para as seguintes situações:

a. Utilize uma transação para incluir uma venda para cada vendedor cadastrado. Caso ocorra erro de SQL em alguma dessas inclusões, realize **ROLLBACK**, ajuste o *script* e execute novamente. Ao final, persista utilizando **COMMIT**.

b. Considerando três registros (linhas) em vez de tabelas, provoque uma situação de *deadlock* como a da figura 9, utilizando transações em três conexões diferentes.