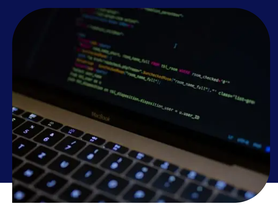
BLOG

Understanding Deadlocks in MySQL & PostgreSQL

Sebastian Insausti
Published May 21, 2018 · Performance Management MySQL PostgreSQL



Working with databases, concurrency control is the concept that ensures that database transactions are performed concurrently without violating data integrity.

There is a lot of theory and different approaches around this concept and how to accomplish it, but we will briefly refer to the way that PostgreSQL and MySQL (when using InnoDB) handle it, and a common problem that can arise in highly concurrent systems: deadlocks.

These engines implement concurrency control by using a method called MVCC (Multiversion Concurrency Control). In this method, when an item is being updated, the changes will not overwrite the original data, but instead, a new version of the item (with the changes) will be created. Thus we will have several versions of the item stored.

One of the main advantages of this model is that locks acquired for querying (reading) data do not conflict with locks acquired for writing data, and so reading never blocks writing, and writing never blocks reading.

But, if several versions of the same item are stored, which version of it will a transaction see? To answer that question we need to review the concept of transaction isolation. Transactions specify an isolation level, that defines the degree to which one transaction must be isolated from resource or data modifications made by other transactions. This degree is directly related with the locking generated by a transaction, and so, as it can be specified at transaction level, it can determine the impact that a running transaction can have over other running transactions.

This is a very interesting and long topic, although we will not go into too many details in this blog. We'd recommend the PostgreSQL and MySQL official documentation for further reading on this topic.

So, why are we going into the above topics when dealing with deadlocks? Because sql commands will automatically acquire locks to ensure the MVCC behavior, and the lock type acquired depends on the transaction isolation defined.

There are several types of locks (again another long and interesting topic to review for PostgreSQL and MySQL) but, the important thing about them, is how they interact (most exactly, how they conflict) with each other. Why is that? Because two transactions cannot hold locks of conflicting modes on the same object at the same time. And a non-minor detail, once acquired, a lock is normally held till end of the transaction.

This is a PostgreSQL example of how locking types conflict with each other:

Requested Lock Mode	Current Lock Mode							
	ACCESS SHARE	ROW SHARE	ROW EXCLUSIVE	SHARE UPDATE EXCLUSIVE	SHARE	SHARE ROW EXCLUSIVE	EXCLUSIVE	ACCESS EXCLUSIVE
ACCESS SHARE								X
ROW SHARE							X	X
ROW EXCLUSIVE					X	X	X	X
SHARE UPDATE EXCLUSIVE				X	X	X	X	X
SHARE			X	X		X	X	X
SHARE ROW EXCLUSIVE			X	X	X	X	X	X
EXCLUSIVE		X	X	X	X	X	X	X
ACCESS EXCLUSIVE	X	X	X	X	X	X	X	X

PostgreSQL Locking types conflict

And for MySQL:

	X	IX	S	IS
X	Conflict	Conflict	Conflict	Conflict
IX	Conflict	Compatible	Conflict	Compatible
S	Conflict	Conflict	Compatible	Compatible
IS	Conflict	Compatible	Compatible	Compatible

MySQL Locking types conflict
X= exclusive lock IX= intention exclusive lock S= shared lock IS= intention shared lock

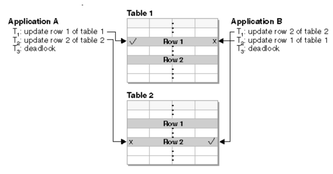
So what happens when I have two running transactions that want to hold conflicting locks on the same object at the same time? One of them will get the lock and the other will have to wait.

So now we are in a position to truly understand what is happening during a deadlock.

What is a deadlock then? As you can imagine, there are several definitions for a database deadlock, but I like the following for its simplicity.

A database deadlock is a situation in which two or more transactions are waiting for one another to give up locks.

So for example, the following situation will lead us to a deadlock:



Deadlock example

Here, application A gets a lock on table 1 row 1 in order to make an update.

At the same time application B gets a lock on table 2 row 2.

Now application A needs to get a lock on table 2 row 2, in order to continue the

execution and finish the transaction, but it cannot get the lock because it is held by application B. Application A needs to wait for application B to release it.

Application B needs to get a lock on table 1 row 1 in order to continue the execution and finish the transaction, but it cannot get the lock because it is held by application A.

So here we are in a deadlock situation. Application A is waiting for the resource held by application B in order to finish and application B is waiting for the resource held by application A. So, how to continue? The database engine will detect the deadlock and kill one of the transactions, unblocking the other one and raising a deadlock error on the killed one.

Let's check some PostgreSQL and MySQL deadlock examples:

PostgreSQL

Suppose we have a test database with information from the countries of the world.

```
world=# SELECT code,region,population FROM country WHERE code IN ('NLD','A
code |          region            | population
------+----------------------------+------------
NLD  | Western Europe             |   15864000
AUS  | Australia and New Zealand  |   18886000
(2 rows)
```

We have two sessions that want to make changes to the database.

The first session will modify the region field for the NLD code, and the population field for the AUS code.

The second session will modify the region field for the AUS code, and the population field for the NLD code.

Table data:

```
code: NLD
region: Western Europe
population: 15864000
```

```
code: AUS
region: Australia and New Zealand
population: 18886000
```

Session 1:

```
world=# BEGIN;
BEGIN
world=# UPDATE country SET region='Europe' WHERE code='NLD';
UPDATE 1
```

Session 2:

```
world=# BEGIN;
BEGIN
world=# UPDATE country SET region='Oceania' WHERE code='AUS';
UPDATE 1
world=# UPDATE country SET population=15864001 WHERE code='NLD';
```

Session 2 will hang waiting for Session 1 to finish.

Session 1:

```
world=# UPDATE country SET population=18886001 WHERE code='AUS';

ERROR:  deadlock detected
DETAIL:  Process 1181 waits for ShareLock on transaction 579; blocked by p
Process 1148 waits for ShareLock on transaction 578; blocked by process 11
HINT:  See server log for query details.
CONTEXT:  while updating tuple (0,15) in relation "country"
```

Here we have our deadlock. The system detected the deadlock and killed session 1.

Session 2:

```
world=# BEGIN;
BEGIN
world=# UPDATE country SET region='Oceania' WHERE code='AUS';
UPDATE 1
world=# UPDATE country SET population=15864001 WHERE code='NLD';
UPDATE 1
```

And we can check that the second session finished correctly after the deadlock was detected and the Session 1 was killed (thus, the lock was released).

To have more details we can see the log in our PostgreSQL server:

```
2018-05-16 12:56:38.520 -03 [1181] ERROR:  deadlock detected
2018-05-16 12:56:38.520 -03 [1181] DETAIL:  Process 1181 waits for ShareLo
       Process 1148 waits for ShareLock on transaction 578; blocked by pro
       Process 1181: UPDATE country SET population=18886001 WHERE code='AU
       Process 1148: UPDATE country SET population=15864001 WHERE code='NL
2018-05-16 12:56:38.520 -03 [1181] HINT:  See server log for query details
2018-05-16 12:56:38.520 -03 [1181] CONTEXT:  while updating tuple (0,15) i
2018-05-16 12:56:38.520 -03 [1181] STATEMENT:  UPDATE country SET populati
2018-05-16 12:59:50.568 -03 [1181] ERROR:  current transaction is aborted,
```

Here we will be able to see the actual commands that were detected on deadlock.

MySQL

To simulate a deadlock in MySQL we can do the following.

As with PostgreSQL, suppose we have a test database with information on actors and movies among other things.

```
mysql> SELECT first_name,last_name FROM actor WHERE actor_id IN (1,7);
```

```
+------------+-----------+
| first_name | last_name |
+------------+-----------+
| PENELOPE   | GUINESS   |
| GRACE      | MOSTEL    |
+------------+-----------+
2 rows in set (0.00 sec)
```

We have two processes that want to make changes to the database.

The first process will modify the field first_name for actor_id 1, and the field last_name for actor_id 7.

The second process will modify the field first_name for actor_id 7, and the field last_name for actor_id 1.

Table data:

```
actor_id: 1
first_name: PENELOPE
last_name: GUINESS
```

```
actor_id: 7
first_name: GRACE
last_name: MOSTEL
```

Session 1:

```
mysql> set autocommit=0;
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> BEGIN;
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> UPDATE actor SET first_name='GUINESS' WHERE actor_id='1';
Query OK, 1 row affected (0.01 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

Session 2:

```
mysql> set autocommit=0;
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> BEGIN;
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> UPDATE actor SET first_name='MOSTEL' WHERE actor_id='7';
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

```
mysql> UPDATE actor SET last_name='PENELOPE' WHERE actor_id='1';
```

Session 2 will hang waiting for Session 1 to finish.

Session 1:

```
mysql> UPDATE actor SET last_name='GRACE' WHERE actor_id='7';

ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting
```

Here we have our deadlock. The system detected the deadlock and killed session 1.

Session 2:

```
mysql> set autocommit=0;
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> BEGIN;
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> UPDATE actor SET first_name='MOSTEL' WHERE actor_id='7';
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

```
mysql> UPDATE actor SET last_name='PENELOPE' WHERE actor_id='1';
Query OK, 1 row affected (8.52 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

As we can see in the error, as we saw for PostgreSQL, there is a deadlock between both processes.

For more details we can use the command SHOW ENGINE INNODB STATUSG:

```
mysql> SHOW ENGINE INNODB STATUSG
------------------------
LATEST DETECTED DEADLOCK
------------------------
2018-05-16 18:55:46 0x7f4c34128700
*** (1) TRANSACTION:
TRANSACTION 1456, ACTIVE 33 sec starting index read
mysql tables in use 1, locked 1
LOCK WAIT 3 lock struct(s), heap size 1136, 2 row lock(s), undo log entrie
MySQL thread id 54, OS thread handle 139965388506880, query id 15876 local
UPDATE actor SET last_name='PENELOPE' WHERE actor_id='1'
*** (1) WAITING FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 23 page no 3 n bits 272 index PRIMARY of table `saki
Record lock, heap no 2 PHYSICAL RECORD: n_fields 6; compact format; info b
0: len 2; hex 0001; asc   ;;
1: len 6; hex 0000000005af; asc      ;;
2: len 7; hex 2d000001690110; asc -   i  ;;
3: len 7; hex 4755494e455353; asc GUINESS;;
```

```
4: len 7; hex 4755494e455353; asc GUINESS;;
5: len 4; hex 5afca8b3; asc Z   ;;

*** (2) TRANSACTION:
TRANSACTION 1455, ACTIVE 47 sec starting index read, thread declared insic
mysql tables in use 1, locked 1
3 lock struct(s), heap size 1136, 2 row lock(s), undo log entries 1
MySQL thread id 53, OS thread handle 139965267871488, query id 16013 local
UPDATE actor SET last_name='GRACE' WHERE actor_id='7'
*** (2) HOLDS THE LOCK(S):
RECORD LOCKS space id 23 page no 3 n bits 272 index PRIMARY of table `saki
Record lock, heap no 2 PHYSICAL RECORD: n_fields 6; compact format; info b
0: len 2; hex 0001; asc   ;;
1: len 6; hex 0000000005af; asc      ;;
2: len 7; hex 2d000001690110; asc -   i  ;;
3: len 7; hex 4755494e455353; asc GUINESS;;
4: len 7; hex 4755494e455353; asc GUINESS;;
5: len 4; hex 5afca8b3; asc Z   ;;

*** (2) WAITING FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 23 page no 3 n bits 272 index PRIMARY of table `saki
Record lock, heap no 202 PHYSICAL RECORD: n_fields 6; compact format; infc
0: len 2; hex 0007; asc   ;;
1: len 6; hex 0000000005b0; asc      ;;
2: len 7; hex 2e0000016a0110; asc .   j  ;;
3: len 6; hex 4d4f5354454c; asc MOSTEL;;
4: len 6; hex 4d4f5354454c; asc MOSTEL;;
5: len 4; hex 5afca8c1; asc Z   ;;

*** WE ROLL BACK TRANSACTION (2)
```

Under the title "LATEST DETECTED DEADLOCK", we can see details of our deadlock.

To see the detail of the deadlock in the mysql error log, we must enable the option innodb_print_all_deadlocks in our database.

```
mysql> set global innodb_print_all_deadlocks=1;
Query OK, 0 rows affected (0.00 sec)
```

MySQL Log Error:

```
2018-05-17T18:36:58.341835Z 12 [Note] InnoDB: Transactions deadlock detect
2018-05-17T18:36:58.341869Z 12 [Note] InnoDB:
*** (1) TRANSACTION:

TRANSACTION 1812, ACTIVE 42 sec starting index read
mysql tables in use 1, locked 1
LOCK WAIT 3 lock struct(s), heap size 1136, 2 row lock(s), undo log entrie
MySQL thread id 11, OS thread handle 140515492943616, query id 8467 localh
UPDATE actor SET last_name='PENELOPE' WHERE actor_id='1'
2018-05-17T18:36:58.341945Z 12 [Note] InnoDB: *** (1) WAITING FOR THIS LOC

RECORD LOCKS space id 23 page no 3 n bits 272 index PRIMARY of table `saki
Record lock, heap no 204 PHYSICAL RECORD: n_fields 6; compact format; infc
0: len 2; hex 0001; asc   ;;
1: len 6; hex 000000000713; asc      ;;
2: len 7; hex 330000016b0110; asc 3   k  ;;
3: len 7; hex 4755494e455353; asc GUINESS;;
4: len 7; hex 4755494e455353; asc GUINESS;;
5: len 4; hex 5afdcb89; asc Z   ;;

2018-05-17T18:36:58.342347Z 12 [Note] InnoDB: *** (2) TRANSACTION:

TRANSACTION 1811, ACTIVE 65 sec starting index read, thread declared insic
mysql tables in use 1, locked 1
3 lock struct(s), heap size 1136, 2 row lock(s), undo log entries 1
MySQL thread id 12, OS thread handle 140515492677376, query id 9075 localh
UPDATE actor SET last_name='GRACE' WHERE actor_id='7'
2018-05-17T18:36:58.342409Z 12 [Note] InnoDB: *** (2) HOLDS THE LOCK(S):

RECORD LOCKS space id 23 page no 3 n bits 272 index PRIMARY of table `saki
Record lock, heap no 204 PHYSICAL RECORD: n_fields 6; compact format; infc
0: len 2; hex 0001; asc   ;;
1: len 6; hex 000000000713; asc      ;;
2: len 7; hex 330000016b0110; asc 3   k  ;;
3: len 7; hex 4755494e455353; asc GUINESS;;
4: len 7; hex 4755494e455353; asc GUINESS;;
5: len 4; hex 5afdcb89; asc Z   ;;

2018-05-17T18:36:58.342793Z 12 [Note] InnoDB: *** (2) WAITING FOR THIS LOC

RECORD LOCKS space id 23 page no 3 n bits 272 index PRIMARY of table `saki
Record lock, heap no 205 PHYSICAL RECORD: n_fields 6; compact format; infc
0: len 2; hex 0007; asc   ;;
1: len 6; hex 000000000714; asc      ;;
2: len 7; hex 340000016c0110; asc 4   l  ;;
3: len 6; hex 4d4f5354454c; asc MOSTEL;;
4: len 6; hex 4d4f5354454c; asc MOSTEL;;
5: len 4; hex 5afdcba0; asc Z   ;;

2018-05-17T18:36:58.343105Z 12 [Note] InnoDB: *** WE ROLL BACK TRANSACTION
```

Taking into account what we have learned above about why deadlocks happen, you can see that there is not much we can do on the database side to avoid them. Anyway, as DBAs it is our duty to actually catch them, analyze them, and provide feedback to the developers.

The reality is that these errors are particular to each application, so you will need to check them one by one and there is not guide to tell you how to troubleshoot this. Keeping this in mind, there are some things you can look for.

Tips for investigating and avoiding deadlocks

Search for long-running transactions. As the locks are usually held until the end of a transaction, the longer the transaction, the longer the locks over the resources. If it is possible, try to split long-running transactions into smaller/faster ones.

Sometimes it is not possible to actually split the transactions, so the work should focus on trying to execute those operations in a consistent order each time, so transactions form well-defined queues and do not deadlock.

One workaround that you can also propose is to add retry logic into the application (of course, try to solve the underlying issue first) in a way that, if a deadlock happens, the application will run the same commands again.

Check the isolation levels used, sometimes you try by changing them. Look for commands like SELECT FOR UPDATE, and SELECT FOR SHARE, as they generate explicit locks, and evaluate if they are really needed or you can work with an older snapshot of the data. One thing you can try if you cannot remove these commands is using a lower isolation level such as READ COMMITTED.

Of course, always add well-chosen indexes to your tables. Then your queries need to scan fewer index records and consequently set fewer locks.

On a higher level, as a DBA you can take some precautions to minimize locking in general. To name one example, in this case for PostgreSQL, you can avoid adding a default value in the same command that you will add a column. Altering a table will get a really aggressive lock, and setting a default value for it will actually update the existing rows that have null values, making this operation take really long. So if you split this operation into several commands, adding the column, adding the default, updating the null values, you will minimize the locking impact.

Of course, there are tons of tips like this that the DBAs get with the practice (creating indexes concurrently, creating the pk index separately before adding the pk, and so on), but the important thing is to learn and understand this "way of thinking" and always to minimize the lock impact of the operations we are doing.

Summary

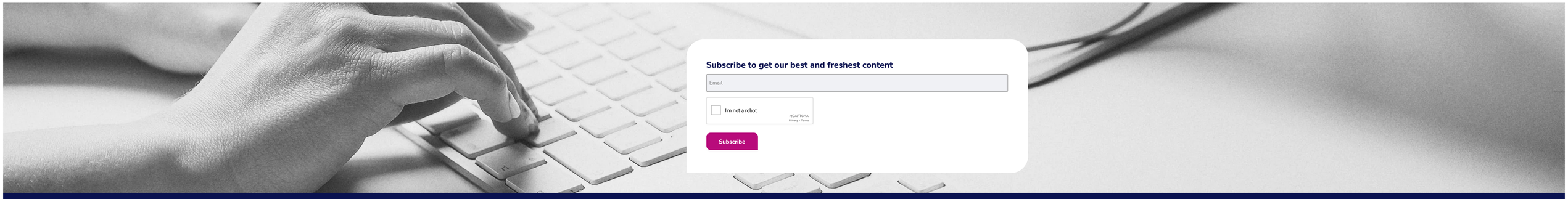
Hopefully, this blog has provided you with helpful information on database deadlocks and how to overcome them. Since there isn't a sure-fire way to avoid deadlocks, knowing how they work can help you catch them before they do harm to your database instances. Software solutions like ClusterControl can help you ensure that your databases always stay in shape. ClusterControl has already helped hundreds of enterprises – will yours be next? Download your free trial of ClusterControl today to see if it's the right fit for your database needs.

Related content

September 8, 2021 Lukas Vileikis
ClusterControl supports Redis with version 1.9

September 7, 2021 Lukas Vileikis
Your DBA is Absent: What Do You Do?

September 3, 2021 Lukas Vileikis
What's the Use of Developer Studio in ClusterControl?



Subscribe to get our best and freshest content

Email

I'm not a robot reCAPTCHA Privacy - Terms

Subscribe

Products
CC
CCX
Backup Ninja

CC
MySQL
MariaDB
MongoDB
PostgreSQL
MySQL NDB
TimescaleDB
MySQL Galera

Solutions
Multi cloud
Hybrid cloud
Disaster recovery
Cloud Service Providers (CSPs)

Developers
Docs
Support
S9S slack
CC Training
CC Certification

Corporate
About us
Contact
Careers
Partners

The Planet9s newsletter

Join for inspiration, news about database stuff, this, that and more.

Email

I'm not a robot reCAPTCHA Privacy - Terms

Subscribe