

DEADLOCK DETECTION MECHANISM IN POSTGRES

TABLE OF CONTENTS

- What is DeadLock?
- Detection/Correction Algorithm
- How Algorithm is implemented in Practical scenario
- Conclusion

What is Deadlock?

Deadlock, as seen in Operating system concepts, is the condition in which one process, say A, wants a resource which is held by the other process B. B wants a resource from A to finish its execution. Since both the processes will not release the resources it holds until its execution is complete, they keep waiting for the resources infinitely. Thus causing a deadlock.

A similar situation can arise in database system as well. Instead of resources, the transaction processes keep waiting for 'locks' for an object from one another.

This can be well explained with an example.

Consider two transactions trying to modify the same table. In transaction 1 we have something like this..

```
UPDATE student SET marks = marks + 10.00  
WHERE usn = 123;
```

As this statement successfully executes, transaction 1 acquires something called Row-level lock for the row with usn = 123. Basically what this means is that since Transaction 1 has modified this row but has not committed yet, let's lock up this row so that no other transaction can see this modification.

Now in another transaction 2 we do some modification like these..

```
UPDATE student SET marks = marks + 10.00  
WHERE usn = 120;
```

```
UPDATE student SET marks = marks + 10.00  
WHERE usn = 123;
```

The first statement updates correctly and modifies the value. It also obtains the lock for that row. Next it tries to update the row with usn=123. But finds that the lock for that row is with Transaction 1. So it waits patiently. Now to make matters worse, we do something like this from Transaction 1

```
UPDATE student SET marks = marks + 10.00  
WHERE usn = 120;
```

Now since the row lock of usn=120 belongs to transaction 2, transaction 1 waits for transaction 2 to complete.

So ultimately, each transaction is waiting for the other to complete and release the locks. This waiting is infinite and hence a deadlock .

So, what do we do when we come across a deadlock? The solution is fairly simple. We abort any one of the transaction, rollback the changes and let the other commit.

But which transaction should be aborted?

What if there are multiple deadlocks?

What is aborting a transaction causes another deadlock?

To solve a similar bunch of complex questions, Postgres uses a Deadlock Detection Algorithm.

Deadlock detection algorithm used in Postgres

A special consideration to be noted:

- A key design consideration is that we want to make routine operations (lock grant and release) run quickly when there is no deadlock, and avoid the overhead of deadlock handling as much as possible.

This is done as follows.

'If a process cannot acquire the lock it wants immediately, it goes to sleep without any deadlock check. But it also sets a delay timer. If the delay expires before the process is granted the lock it wants, it runs the deadlock detection/breaking code. Normally this code will determine that there is no deadlock condition, and then the process will go back to sleep and wait quietly until it is granted the lock. But if a deadlock condition does exist, it will be resolved, usually by aborting the detecting process transaction.'

These rules are followed in granting locks:

1. A lock request is granted immediately if it does not conflict with any existing or waiting lock request, or if the process already holds an

instance of the same lock type. Note that a process never conflicts with itself, eg one can obtain read lock when one already holds exclusive lock(update lock).

2. Otherwise the process joins the lock's wait queue. Normally it will be added to the end of the queue, but there is an exception: if the process already holds locks on this same lockable object that conflict with the request of any pending waiter, then the process will be inserted in the wait queue just ahead of the first such waiter.(This extra check is always better than a future deadlock condition.)

3. When inserting before the end of the queue: if the process's request does not conflict with any existing lock nor any waiting request before its insertion point, then go ahead and grant the lock without waiting.

Now, when a lock is released, the lock release routine (**ProcLockWakeup**) scans the lock object's wait queue. Each waiter is awoken if

(a) its request does not conflict with already-granted locks, and

(b) its request does not conflict with the requests of prior un-wakable waiters.

Rule (b) ensures that conflicting requests are granted in order of arrival. There are cases where a later waiter must be allowed to go in front of conflicting earlier waiters to avoid deadlock, but it is not ProcLockWakeup's responsibility to recognize these cases; instead, the deadlock detection code will re-order the wait queue when necessary.

To perform deadlock checking, we use the standard method of viewing the various processes as nodes in a directed graph (the **waits-for graph** or **WFG**). There is a graph edge leading from process A to process B if A waits for B, ie, A is waiting for some lock and B holds a conflicting lock. There is a deadlock condition if and only if the WFG contains a cycle. We detect cycles by searching outward along waits-for edges to see if we return to our starting point.

There are three possible outcomes:

1. All outgoing paths terminate at a running process (which has no outgoing edge).

2. A deadlock is detected by looping back to the start point. We resolve such a deadlock by canceling the start point's lock request and reporting an error in that transaction, which normally leads to transaction abort and release of that transaction's held locks. Note that it's sufficient to cancel one request to remove the cycle, we don't need to kill all the transactions involved.

3. Some path(s) loop back to a node other than the start point. This indicates a deadlock, but one that does not involve our starting process. We ignore this condition on the grounds that resolving such a deadlock is the responsibility of the processes involved --- killing our start-point process would not resolve the deadlock. So, cases 1 and 3 both report "no deadlock".

Postgres' situation is a little more complex than the standard discussion of deadlock detection, for two reasons:

1. A process can be waiting for more than one other process, since there might be multiple deadlock situation. This creates no real difficulty however, we simply need to be prepared to trace more than one outgoing edge.

2. If a process A is behind a process B in some lock's wait queue, and their requested locks conflict, then we must say that A waits for B, since **ProcLockWakeup** will never awaken A before B. This creates additional edges in the WFG. We call these "soft" edges, as opposed to the "hard" edges induced by locks already held.

A "soft" block, or wait-priority block, has the same potential for inducing deadlock as a hard block. However, we may be able to resolve a soft block without aborting the transactions involved: we can instead rearrange the order of the wait queue. This rearrangement reverses the direction of the soft edge between two processes with conflicting requests whose queue order is reversed. If we can find a rearrangement that eliminates a cycle without creating new ones, then we can avoid an abort. Checking for such possible rearrangements is the trickiest part of the algorithm.

The major deadlock detector is a routine **FindLockCycle()** which is given a starting point process. It recursively scans outward across waits-for edges as discussed above. If it finds no cycle involving the start point, it returns "false".

When such a cycle is found, `FindLockCycle()` returns "true", and as it unwinds it also builds a list of any "soft" edges involved in the cycle. If the resulting list is empty then there is a hard deadlock and the configuration cannot succeed. However, if the list is not empty, then reversing any one of the listed edges through wait-queue rearrangement will eliminate that cycle. Since such a reversal might create cycles elsewhere, we may need to try every possibility.

There are many other ways in which this is achieved much efficiently in Postgres and as been explained clearly in the source code for deadlock mechanism.

How Algorithm is implemented in Practical scenario

The example shown below has been implemented on Ubuntu 14.04 LTS running PostgreSQL 9.3.5 and the snapshots have been attached for the same.

3 Transactions have been started which connects to the same database 'student'
Lets call them Ta,Tb, and Tc.

```
Terminal
akarsh@akarsh-Inspiron-3521: ~/Downloads/postgresql-9.3.5/pgsql
akarsh@akarsh-Inspiron-3521:~/Downloads/postgresql-9.3.5/pgsql$ ./bin/psql student
psql (9.3.5)
Type "help" for help.

student=# select * from deadlock ;
 name | value
-----+-----
 B    |    20
 A    |    20
 C    |    20
(3 rows)

student=# begin;
BEGIN
student=# select txid_current();
 txid_current
-----
          929
(1 row)

student=# select pg_backend_pid();
 pg_backend_pid
-----
         11022
(1 row)

student=#
```

```
akarsh@akarsh-Inspiron-3521:~/Downloads/postgresql-9.3.5/pgsql$ ./bin/psql student
psql (9.3.5)
Type "help" for help.

student=# begin;
BEGIN
student=# select txid_current();
 txid_current
-----
          930
(1 row)

student=# select pg_backend_pid();
 pg_backend_pid
-----
         11027
(1 row)

student=#
```

```
akarsh@akarsh-Inspiron-3521:~/Downloads/postgresql-9.3.5/pgsql$ ./bin/psql student
psql (9.3.5)
Type "help" for help.

student=# begin;
BEGIN
student=# select txid_current();
 txid_current
-----
          931
(1 row)

student=# select pg_backend_pid();
 pg_backend_pid
-----
         11033
(1 row)

student=#
```

Thus,

Ta : xid=929 pid=11022

Tb : xid=930 pid=11027

Tc : xid=931 pid=11033

```
Terminal
akarsh@akarsh-Inspiron-3521: ~/Downloads/postgresql-9.3.5/pgsql
akarsh@akarsh-Inspiron-3521:~/Downloads/postgresql-9.3.5/pgsql$ ./bin/psql student
psql (9.3.5)
Type "help" for help.

student=# select * from deadlock ;
 name | value
-----+-----
 B    |    20
 A    |    20
 C    |    20
(3 rows)

student=# begin;
BEGIN
student=# select txid_current();
 txid_current
-----
          929
(1 row)

student=# select pg_backend_pid();
 pg_backend_pid
-----
         11022
(1 row)

student=# update deadlock set value=100 where name='A';
UPDATE 1
student=# update deadlock set value=100 where name='C';
UPDATE 1
student=#
```

```
akarsh@akarsh-Inspiron-3521:~/Downloads/postgresql-9.3.5/pgsql$ ./bin/psql student
psql (9.3.5)
Type "help" for help.

student=# begin;
BEGIN
student=# select txid_current();
 txid_current
-----
          930
(1 row)

student=# select pg_backend_pid();
 pg_backend_pid
-----
         11027
(1 row)

student=# update deadlock set value=100 where name='B';
UPDATE 1
student=# update deadlock set value=100 where name='A';
UPDATE 1
student=#
```

```
akarsh@akarsh-Inspiron-3521:~/Downloads/postgresql-9.3.5/pgsql$ ./bin/psql student
psql (9.3.5)
Type "help" for help.

student=# begin;
BEGIN
student=# select txid_current();
 txid_current
-----
          931
(1 row)

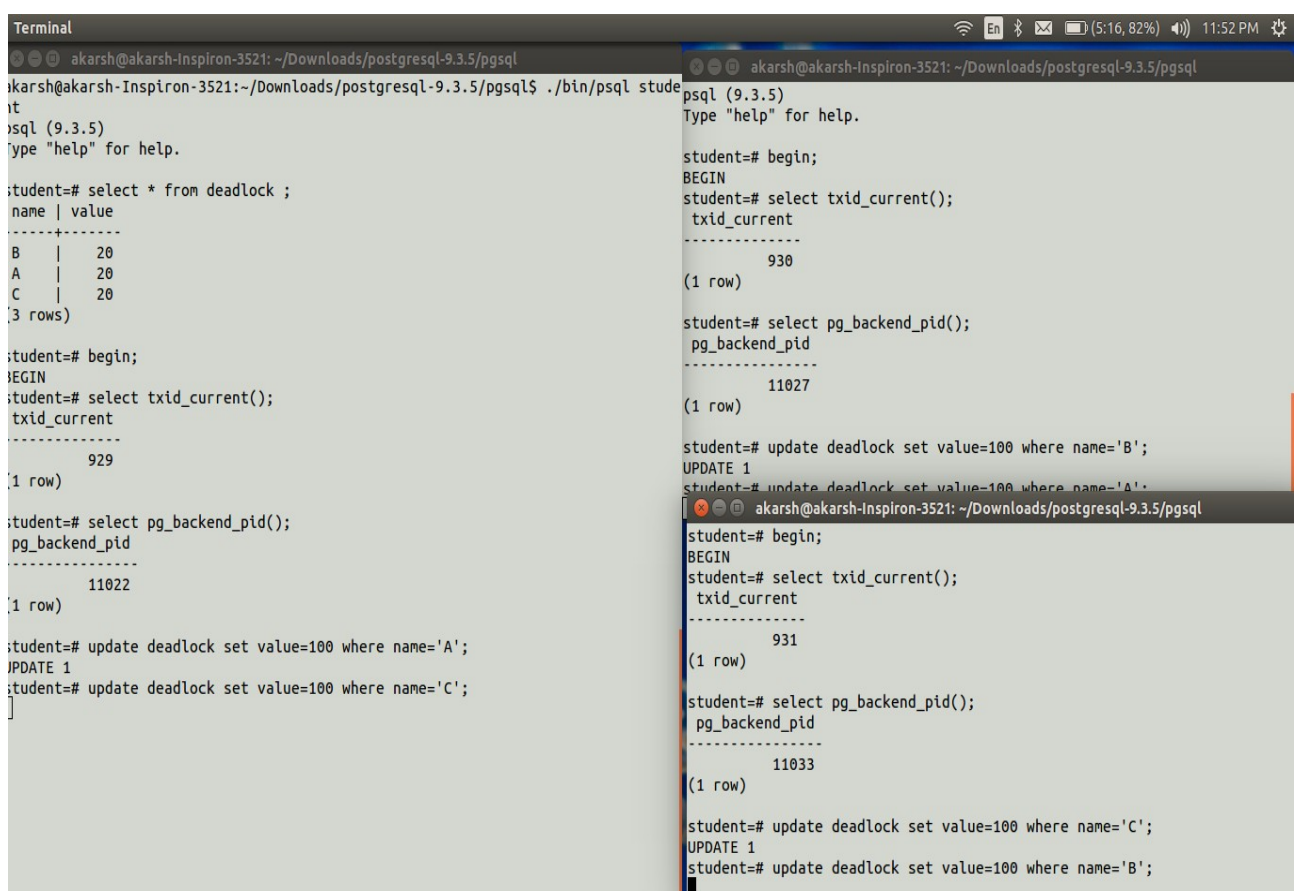
student=# select pg_backend_pid();
 pg_backend_pid
-----
         11033
(1 row)

student=# update deadlock set value=100 where name='C';
UPDATE 1
student=#
```

Ta successfully updates A and obtains its lock.

Tb successfully updates B and obtains its lock. It tries to update A but since its lock is not available it waits.

Tc successfully updates C and obtains its lock.



The image shows three terminal windows illustrating a deadlock scenario in PostgreSQL. The leftmost window shows the initial state of the 'deadlock' table and the execution of a transaction (Ta) that updates row A. The middle window shows transaction (Tb) updating row B and then attempting to update row A, where it must wait because row A is locked by Ta. The rightmost window shows transaction (Tc) updating row C. The sequence of operations leads to a deadlock where Ta is waiting for Tc to finish, and Tc is waiting for Tb to finish, which is waiting for Ta.

```
Terminal
akarsh@akarsh-Inspiron-3521: ~/Downloads/postgresql-9.3.5/pgsql
akarsh@akarsh-Inspiron-3521:~/Downloads/postgresql-9.3.5/pgsql$ ./bin/psql student
psql (9.3.5)
Type "help" for help.

student=# select * from deadlock ;
 name | value 
-----+-----
 B    |    20 
 A    |    20 
 C    |    20 
(3 rows)

student=# begin;
BEGIN
student=# select txid_current();
 txid_current 
-----
          929 
(1 row)

student=# select pg_backend_pid();
 pg_backend_pid 
-----
          11022 
(1 row)

student=# update deadlock set value=100 where name='A';
UPDATE 1
student=# update deadlock set value=100 where name='C';
[

psql (9.3.5)
Type "help" for help.

student=# begin;
BEGIN
student=# select txid_current();
 txid_current 
-----
          930 
(1 row)

student=# select pg_backend_pid();
 pg_backend_pid 
-----
          11027 
(1 row)

student=# update deadlock set value=100 where name='B';
UPDATE 1
student=# update deadlock set value=100 where name='A';
[

akarsh@akarsh-Inspiron-3521: ~/Downloads/postgresql-9.3.5/pgsql
student=# begin;
BEGIN
student=# select txid_current();
 txid_current 
-----
          931 
(1 row)

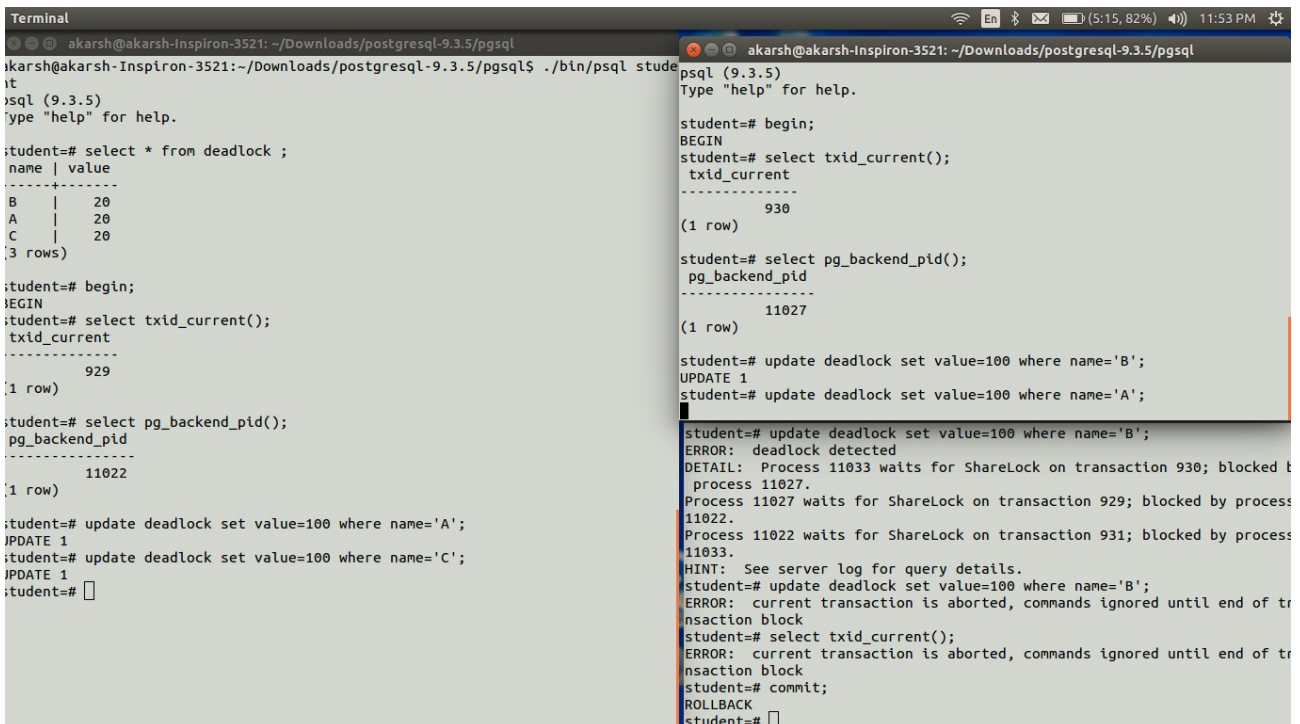
student=# select pg_backend_pid();
 pg_backend_pid 
-----
          11033 
(1 row)

student=# update deadlock set value=100 where name='C';
UPDATE 1
student=# update deadlock set value=100 where name='B';
[
```

Now, Ta tries to update C but since its locked it waits for Tc. Tc tries to update B which is locked by Tb. Hence we reach a deadlock.

Tc->Tb->Ta->Tc

According to our algorithm, Tc must be aborted.



The image shows two terminal windows side-by-side, both running PostgreSQL 9.3.5. The left window shows a sequence of queries: a SELECT from 'deadlock' showing three rows (B, A, C) with value 20; a BEGIN transaction; a SELECT of txid_current() returning 929; a SELECT of pg_backend_pid() returning 11022; and three UPDATE statements on the 'deadlock' table for names 'A', 'C', and 'B', each with a value of 100. The right window shows a similar sequence: a SELECT of txid_current() returning 930; a SELECT of pg_backend_pid() returning 11027; and UPDATE statements for 'B', 'A', and 'B'. After the second 'B' update, an error message appears: 'ERROR: deadlock detected'. The message continues: 'DETAIL: Process 11033 waits for ShareLock on transaction 930; blocked by process 11027. Process 11027 waits for ShareLock on transaction 929; blocked by process 11022. Process 11022 waits for ShareLock on transaction 931; blocked by process 11033. HINT: See server log for query details.' This indicates a deadlock between the two transactions.

```
Terminal
akarsh@akarsh-Inspiron-3521: ~/Downloads/postgresql-9.3.5/pgsql
akarsh@akarsh-Inspiron-3521:~/Downloads/postgresql-9.3.5/pgsql$ ./bin/psql student
psql (9.3.5)
Type "help" for help.

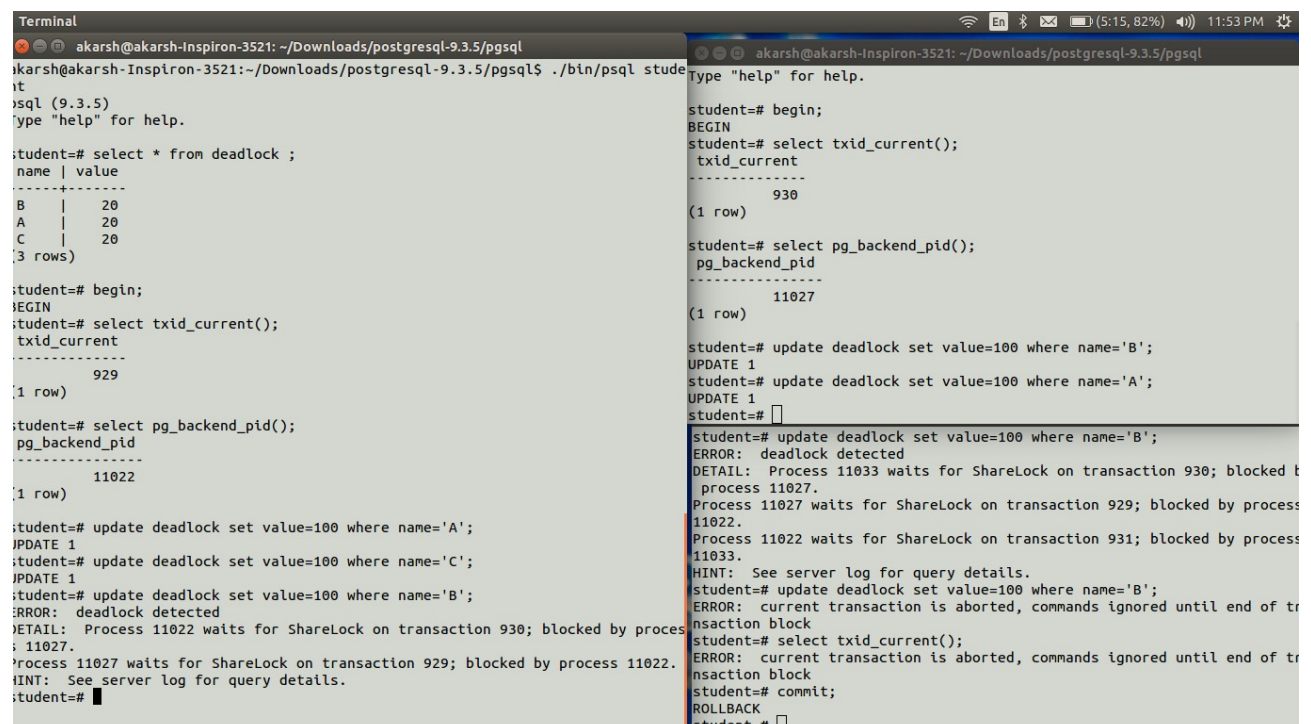
student=# select * from deadlock ;
 name | value 
-----+-----
 B    |    20 
 A    |    20 
 C    |    20 
(3 rows)

student=# begin;
BEGIN
student=# select txid_current();
 txid_current 
-----
          929 
(1 row)

student=# select pg_backend_pid();
 pg_backend_pid 
-----
          11022 
(1 row)

student=# update deadlock set value=100 where name='A';
UPDATE 1
student=# update deadlock set value=100 where name='C';
UPDATE 1
student=# update deadlock set value=100 where name='B';
ERROR: deadlock detected
DETAIL:  Process 11033 waits for ShareLock on transaction 930; blocked by process 11027.
Process 11027 waits for ShareLock on transaction 929; blocked by process 11022.
Process 11022 waits for ShareLock on transaction 931; blocked by process 11033.
HINT:  See server log for query details.
student=# update deadlock set value=100 where name='B';
ERROR: current transaction is aborted, commands ignored until end of transaction block
student=# select txid_current();
ERROR: current transaction is aborted, commands ignored until end of transaction block
student=# commit;
ROLLBACK
student=#
```

Tc rolls back, Thus the lock for C is released.
Ta successfully updates C. Now, Ta tries to update B and reaches another deadlock.



The image shows two terminal windows side-by-side, both running PostgreSQL 9.3.5. The left window shows a sequence of queries: a SELECT from 'deadlock' showing three rows (B, A, C) with value 20; a BEGIN transaction; a SELECT of txid_current() returning 929; a SELECT of pg_backend_pid() returning 11022; and three UPDATE statements on the 'deadlock' table for names 'A', 'C', and 'B', each with a value of 100. The right window shows a similar sequence: a SELECT of txid_current() returning 930; a SELECT of pg_backend_pid() returning 11027; and UPDATE statements for 'B', 'A', and 'B'. After the second 'B' update, an error message appears: 'ERROR: deadlock detected'. The message continues: 'DETAIL: Process 11033 waits for ShareLock on transaction 930; blocked by process 11027. Process 11027 waits for ShareLock on transaction 929; blocked by process 11022. Process 11022 waits for ShareLock on transaction 931; blocked by process 11033. HINT: See server log for query details.' This indicates a deadlock between the two transactions.

```
Terminal
akarsh@akarsh-Inspiron-3521: ~/Downloads/postgresql-9.3.5/pgsql
akarsh@akarsh-Inspiron-3521:~/Downloads/postgresql-9.3.5/pgsql$ ./bin/psql student
psql (9.3.5)
Type "help" for help.

student=# select * from deadlock ;
 name | value 
-----+-----
 B    |    20 
 A    |    20 
 C    |    20 
(3 rows)

student=# begin;
BEGIN
student=# select txid_current();
 txid_current 
-----
          929 
(1 row)

student=# select pg_backend_pid();
 pg_backend_pid 
-----
          11022 
(1 row)

student=# update deadlock set value=100 where name='A';
UPDATE 1
student=# update deadlock set value=100 where name='C';
UPDATE 1
student=# update deadlock set value=100 where name='B';
ERROR: deadlock detected
DETAIL:  Process 11022 waits for ShareLock on transaction 930; blocked by process 11027.
Process 11027 waits for ShareLock on transaction 929; blocked by process 11022.
HINT:  See server log for query details.
student=#
```


Ta is aborted and releases locks for A. Tb successfully updates the value of A and commits.

```
Terminal
akarsh@akarsh-Inspiron-3521: ~/Downloads/postgresql-9.3.5/pgsql

student=# select * from deadlock ;
name | value
-----+-----
B    |    20
A    |    20
C    |    20
(3 rows)

student=# begin;
BEGIN
student=# select txid_current();
txid_current
-----
929
(1 row)

student=# select pg_backend_pid();
pg_backend_pid
-----
11022
(1 row)

student=# update deadlock set value=100 where name='A';
UPDATE 1
student=# update deadlock set value=100 where name='C';
UPDATE 1
student=# update deadlock set value=100 where name='B';
ERROR:  deadlock detected
DETAIL:  Process 11022 waits for ShareLock on transaction 930; blocked by process 11027.
Process 11027 waits for ShareLock on transaction 929; blocked by process 11022.
Process 11022 waits for ShareLock on transaction 931; blocked by process 11033.
HINT:  See server log for query details.
student=# update deadlock set value=100 where name='B';
ERROR:  current transaction is aborted, commands ignored until end of transaction block
student=# commit;
ROLLBACK
student=#
```

```
akarsh@akarsh-Inspiron-3521: ~/Downloads/postgresql-9.3.5/pgsql

(1 row)

student=# select pg_backend_pid();
pg_backend_pid
-----
11027
(1 row)

student=# update deadlock set value=100 where name='B';
UPDATE 1
student=# update deadlock set value=100 where name='A';
UPDATE 1
student=# select * from deadlock ;
name | value
-----+-----
C    |    20
B    |   100
A    |   100
(3 rows)

student=#
student=# update deadlock set value=100 where name='B';
ERROR:  deadlock detected
DETAIL:  Process 11033 waits for ShareLock on transaction 930; blocked by process 11027.
Process 11027 waits for ShareLock on transaction 929; blocked by process 11022.
Process 11022 waits for ShareLock on transaction 931; blocked by process 11033.
HINT:  See server log for query details.
student=# update deadlock set value=100 where name='B';
ERROR:  current transaction is aborted, commands ignored until end of transaction block
student=# select txid_current();
ERROR:  current transaction is aborted, commands ignored until end of transaction block
student=# commit;
ROLLBACK
student=#
```

Conclusion

The deadlock detection algorithm used by postgres seems complex but fairly essential to maintain its flexibility and concurrency. However there is always scope for a better algorithm.