Notes on concurrency in PostgreSQL

Maurizio Lenzerini

We refer to a database managed by PostgreSQL with the table “persone” defined as follows:

```
CREATE TABLE persone
(
    nome character varying(40),
    eta integer,
    reddito integer,
    CONSTRAINT persona_pkey PRIMARY KEY (nome)
)
```

and with the following tuples:

nome	eta	reddito
Aldo	15	25
Franco	20	60
Luigi	40	50
Luisa	87	75
Maria	42	55
Olga	41	30
Sergio	35	85
Gianna	40	50
Anna	20	50
Filippo	80	26
Andrea	30	27

We remind the reader that a user interacts with the system through sessions. Every command given to the system that is not embraced into an explicit transaction within a session is considered a transaction. If a user wants to define an explicit transaction, the BEGIN command must be given. The transaction is closed by the COMMIT command (or equivalently by the END command), or by the ROLLBACK command. When a transaction is aborted by the system, all the remaining actions are ignored until the END command is issued, at which point all the actions already executed are rollbacked (so, in this case the END command corresponds to ROLLBACK).

If one uses the pgAdmin client, then every session corresponds to a tab. One can always leave a session A and switch to another session B by simply changing tab. In this case, session A

remains active, and the user can always go back to it by accessing the corresponding tab. If the user closes the tab, then the corresponding session expires.

We also remind the reader that the default isolation level in PostgreSQL is READ COMMITTED, which is the first level, since there is no pure "read uncommitted" level in PostgreSQL.

1. Level of isolation: READ-COMMITTED

We start by considering the isolation level READ COMMITTED. That this isolation level is the default in PostgreSQL can be seen by giving the command:

```
show transaction isolation level
```

that causes the system to answer:

```
read committed
```

1.1 Scenario 1: serial schedule

Assume the initial situation of 80 as the value for the "eta" attribute for Filippo. We will open three sessions (tabs in pgAdmin4), each one with one transaction.

Transaction 1
We execute transaction 1, which coincides with the command

```
select * from persone where nome = 'Filippo'
```

In the output we see 80 as the value of "eta", which is the value stored in the database before the begin of the transaction. Since this transaction is just a command, the transaction simply ends immediately.

Transaction 2
We execute transaction 2, which coincides with the command

```
update persone set eta = 50 where nome = 'Filippo'
```

and, since again this is just a command, the corresponding transaction simply ends.

Transaction 3
We execute transaction 3, which coincides with the command

```
select * from persone where nome = 'Filippo'
```

and in the output we see 50 as value of "eta", which is the value stored by transaction 2. Again, since this is just a command, the corresponding transaction simply ends.

Note that this behavior is obvious, since the schedule created by the above-described sequence of actions is serial, and the behavior of the system is the one that everybody expects.

1.2 Scenario 2: the use of multi-version concurrency control

Assume the initial situation of 80 as the value for the "eta" attribute for Filippo. We will open three sessions, each one with one transaction.

Transaction 1
We start executing the following actions of transaction 1:

```
BEGIN;
select * from persone where nome = 'Filippo'
```

and the output shows the value 80 for the "eta" attribute, which is the value stored in the database before the beginning of the transaction. Then we end the transaction by

```
END;
```

Transaction 2
We execute the following actions of transaction 2:

```
BEGIN;
update persone set eta = 50 where nome = 'Filippo'
```

and we leave the session (but we do not close it); therefore the transaction is suspended.

Transaction 3
We execute the following actions of transaction 3:

```
BEGIN;
select * from persone where nome = 'Filippo'
```

and the output shows the value 80 for the "eta" attribute, which is the value stored in the database before the begin of transaction 2. We then end the transaction:

```
END;
```

Note that this means that transaction 2 is working on a local copy (snapshot) of the database. Indeed, the write action of transaction 2 is not yet visible in transaction 3. In other words, transaction 3 sees a version of the Filippo tuple that is different from the one written by transaction 2. We can now go back to the session of transaction 2 and issue the command:

```
END;
```

If we now re-execute Transaction 3 we will see 50 as value of "eta", because transaction 2 has committed (by means of the "END" command), and its effects are now visible to the rest of the transactions. So, the second execution of transaction 3 sees a different version of the Filippo tuple.

1.3 Scenario 3: unrepeatable read

Assume the initial situation of 80 as the value for the "eta" attribute for Filippo. We will open two sessions, each one with one transaction.

Transaction 1
We execute the following actions of transaction 1,

```
BEGIN;
select * from persone where nome = 'Filippo'
```

and the output shows the value 80 for the "eta" attribute, which is the value stored in the database before the begin of the transaction. We leave the session and therefore the transaction is suspended.

Transaction 2
We execute the following actions of transaction 2:

```
BEGIN;
update persone set eta = 50 where nome = 'Filippo';
END;
```

Transaction 1
We execute the following action of transaction 1

```
select * from persone where nome = 'Filippo'
```

and the output shows the value 50 for the "eta" attribute, which is the value stored in the database by transaction 2. We then end the transaction by

```
END;
```

Note that this means that the same transaction (in this case transaction 1) can see different values of the same element in different times. This is clearly a case of the anomaly "unrepeatable read" and is coherent with the isolation level "read committed", that does not enforce "repeatable reads".

2. Level of isolation: REPEATABLE-READ

We now analyze the level of isolation "repeatable-read". In order to enforce this level of isolation in a certain transaction we have to issue the command

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

2.1 Scenario 4: repeatable read

Assume the initial situation of 80 as the value for the "eta" attribute for Filippo. We will open two sessions, each one with one transaction.

Transaction 1
We execute the following actions of transaction 1,

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
select * from persone where nome = 'Filippo'
```

and the output shows the value 80 for the "eta" attribute, which is the value stored in the database before the begin of the transaction. We leave the session and therefore the transaction is suspended. Note that the transaction is defined with the isolation level "repeatable read".

Transaction 2
We execute the following actions of transaction 2:

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
update persone set eta = 50 where nome = 'Filippo'
END;
```

Transaction 1
We execute the following actions of transaction 1

```
select * from persone where nome = 'Filippo'
```
and the output shows the value 80 for the "eta" attribute, which is the value initially stored in the database when transaction 1 started. We then end the transaction by
```
END;
```

Note that this means that the REPEATABLE READ mode starts each transaction with a new snapshot that includes all transactions committed up to that moment. Thus, with REPEATABLE READ, the level of isolation increases. The two read operations within the same transaction 1 see the same data.

Obviously, if we now execute transaction 1 again, then the transaction would see the value 50 for the "eta" attribute for Filippo.

2.2 Scenario 5: the use of locking

Assume the initial situation of 20 as value of the attribute "eta" for Anna, and 80 as value of the attribute "eta" for Filippo. We consider transaction 1 aiming at increasing the "eta" of Filippo, and transaction 2 aiming at decreasing it.

Transaction 1
We execute the following actions of transaction 1:
```
BEGIN;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
update persone set eta = 81 where nome = 'Filippo';
```
and we leave the session and therefore the transaction is suspended.

Transaction 2
We execute the following actions of transaction 2:
```
BEGIN;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
update persone set eta = 79 where nome = Filippo;
```
and what happens at this point is that the system sends a message saying that is waiting for a query to complete. Indeed, transaction 2 is actually put in a queue waiting for transaction 1 releasing the exclusive lock on the tuple relative to Filippo.

Transaction 1
We execute the following actions of transaction 1:
```
END;
```
and the system sends a message saying that the transaction committed.

Transaction 2
If we go back to the session with transaction 2, we see the following message sent by the system:
```
ERROR: could not serialize access due to concurrent update SQL state:
40001
```

and we realize that transaction 2 is aborted. This is because with the isolation level repeatable read, no transaction T can work on data modified by another transaction committed after T started. Note that, if we had ended the transaction 1 with the command

```
ROLLBACK;
```

then transaction 2 would have resumed and the value of the "eta" attribute for Filippo would have modified to 79.

2.3 Scenario 6: deadlock management

Assume the initial situation of 20 as value of the attribute "eta" for Anna, and 80 as value of the attribute "eta" for Filippo. We consider transaction 1 aiming at increasing the "eta" of both Filippo and Anna, and transaction 2 aiming at decreasing them.

Transaction 1
We execute the following actions of transaction 1:

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
update persone set eta = 81 where nome = 'Filippo';
```

and we leave the session and therefore the transaction is suspended.

Transaction 2
We execute the following actions of transaction 2:

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
update persone set eta = 19 where nome = 'Anna';
```

and we leave the session and therefore the transaction is suspended.

Transaction 1
We execute the following actions of transaction 1:

```
update persone set eta = 21 where nome = 'Anna';
```

and the system sends a message saying that is waiting for a query to complete. Indeed, this shows that transaction 1 is actually put in a queue waiting for transaction 2 releasing the exclusive lock on the tuple relative to Anna.

Transaction 2
We execute the following actions of transaction 2:

```
update persone set eta = 79 where nome = 'Filippo';
```

and what happens at this point is that the system aborts transaction 2 and sends the message:

```
ERROR: deadlock detected DETAIL: Process 27605 waits for ShareLock on
transaction 1942; blocked by process 24961. Process 24961 waits for ShareLock
on transaction 1943; blocked by process 27605. HINT: See server log for query
details. SQL state: 40P01
```

and if we end transaction 2 by the command:

```
END
```

the system sends the message:

```
ROLLBACK
```

If we go back to transaction 1, we realize that transaction 1 has proceeded, and this is because it obtained the exclusive lock on the tuple relative to Anna. We can now end transaction 1 by the command:

```
END
```

The database has now the values 81 and 21 for the attribute “eta” for Filippo and Anna, respectively.

This shows that the system adopts a recognition approach to deadlock.