PgDay France 2019

Sécurisez vos transactions concurrentes !

@DanielVerite

A propos de moi

Daniel Vérité -- daniel@manitou-mail.org -- @DanielVerite
Informaticien indépendant, spécialisé bases de données et technos libres.

Communauté des développeurs de PostgreSQL

Proposer, tester, discuter des évolutions et corrections de bugs:
https://www.postgresql.org/list/pgsql-hackers/ et https://www.postgresql.org/list/pgsql-bugs/
Suivre les patches: https://commitfest.postgresql.org

Blog

En français : https://blog-postgresql.verite.pro (https://planete.postgresql.fr)
En anglais : https://postgresql.verite.pro (https://planet.postgresql.org)

Forum

En français : https://forums.postgresql.fr
Q/A en anglais : https://dba.stackexchange.com et https://stackoverflow.com

Projets PostgreSQL:

Appli de mail en base de données: https://www.manitou-mail.org
Séquences aléatoires uniques : https://github.com/dverite/permuteseq
SHAnnn-CRYPT (mots de passe) : https://github.com/dverite/postgres-shacrypt
Chiffrement XTEA/Skip32 : https://github.com/dverite/cryptint
Extension ICU (Unicode avancé): https://github.com/dverite/icu_ext

flexcoin.com

février 2014

flexcoin the bitcoin bank

FAQ Bitcoin Bank Register Activate Login

flexcoin

The World's First Bitcoin Bank

The Solution to a Major Problem

flexcoin fixes one of the biggest problems with the bitcoin currency.

When someone sends you a bitcoin, it is only accessible from the device you initially receive it on. You cannot easily access the funds from your mobile device (or from any other device for that matter). Generally speaking, you must access the bitcoins from the machine they were initially sent to / received on. Using Flexcoin, you can even send bitcoins to an e-mail address. It's designed for anyone to use bitcoins without technical knowledge.

What Flexcoin does is simple - it acts as a central location for all of your bitcoins, and as the world's first bitcoin bank. You can use one account to access all of your bitcoins, from any web connected device.

This gives you the ability to accept a payment at work, and send those same bitcoins to someone else from home that night. Or send them to a friend from your mobile device. Or do anything else with them from any other computer that you have web access on, without first sending it to that device.

This solves multiple problems, and at the same time makes for the first bitcoin bank.

Flexcoin is also a leader in bitcoin security, for example Flexcoin is the first to implement a ZERO link policy in e-mails. No e-mail sent from Flexcoin contains a link or image. If you receive one that does, it's a phishing attempt.

The World's First Bitcoin Bank

In addition to making your bitcoins completely accessible from any web connected device (so you can technically "carry" the bitcoins with you to a store and pay on site), flexcoin becomes the world's first bitcoin bank. Keep reading to find out how.

When you deposit bitcoins with flexcoin, you get a centralized account to send and receive from. You can access them from any web connected device. In addition, we pay discount payments on a positive bitcoin balance at the end of each month.

In other words, if you've got a positive bitcoin balance, you receive a discount payment. We're the world's first bitcoin bank.

Flexcoin is an important part of the bitcoin infrastructure. Our technology allows for instant transfers of bitcoins to a username as compared to next block wait to a huge bitcoin address.

Before Flexcoin: 1555hjPG7pRwTHVMfukPvjXexQMHFE3qu6
After Flexcoin: coffeeshop

Legal Notice: We are not a true bank that accepts USD or any national currency, only bitcoins which by their nature are not regulated, we're not FDIC insured or regulated by any government entity.

flexcoin.com

4 mars 2014

flexcoin the bitcoin bank

FAQ Bitcoin Bank Register Activate Login

Flexcoin is shutting down. (March 3 2014)

On March 2nd 2014 Flexcoin was attacked and robbed of all coins in the hot wallet. The attacker made off with 896 BTC, dividing them into these two addresses:

1NDkevapt4SWYFEmquCDBSf7DLMTNVggdu

1QFcC5JitGwpFKqRDd9QNH3eGN56dCNgy6

As Flexcoin does not have the resources, assets, or otherwise to come back from this loss, we are closing our doors immediately.

Users who put their coins into cold storage will be contacted by Flexcoin and asked to verify their identity. Once identified, cold storage coins will be transferred out free of charge. Cold storage coins were held offline and not within reach of the attacker. All other users will be directed to Flexcoin's "Terms of service" located at "Flexcoin.com/118.html" a document which was agreed on, upon signing up with Flexcoin.

Flexcoin will attempt to work with law enforcement to trace the source of the hack.

Updates will be posted on twitter as soon as they become available.

Update (March 4 2014)

During the investigation into stolen funds we have determined that the extent of the theft was enabled by a flaw within the front-end.

The attacker logged into the flexcoin front end from IP address 207.12.89.117 under a newly created username and deposited to address 1DSD3B3uS2wGZjZAwa2dqQ7M9v7Ajw2iLy

The coins were then left to sit until they had reached 6 confirmations.

The attacker then successfully exploited a flaw in the code which allows transfers between flexcoin users. By sending thousands of simultaneous requests, the attacker was able to "move" coins from one user account to another until the sending account was overdrawn, before balances were updated.

This was then repeated through multiple accounts, snowballing the amount, until the attacker withdrew the coins. (Here and Here)

Flexcoin has made every attempt to keep our servers as secure as possible, including regular testing. In our ~3 years of existence we have successfully repelled thousands of attacks. But in the end, this was simply not enough.

Having this be the demise of our small company, after the endless hours of work we've put in, was never our intent. We've failed our customers, our business, and ultimatley the Bitcoin community.

Please direct any and all questions to admin(at)flexcoin(dot)com and we will reply to you as soon as possible.

The attacker then successfully exploited a **flaw in the code** which allows transfers between flexcoin users.

By sending thousands of simultaneous requests, the attacker was able to "move" coins from one user account to another until the sending account was overdrawn, **before balances were updated**.

Transfert entre portefeuilles

```
CREATE TABLE portefeuille (
    client_id int references client(id),
    ...
    solde numeric,
    ...
) ;
```

```
-- Transfert de montant du portefeuille source_id
-- vers le portefeuille destination_id

BEGIN ;

  SELECT solde FROM portefeuille WHERE client_id = source_id
    INTO v_solde ;

  IF v_solde >= montant THEN
    UPDATE portefeuille SET solde = solde - montant
     WHERE client_id = source_id ;

    UPDATE portefeuille SET solde = solde + montant
     WHERE client_id = destination_id ;

    INSERT INTO historique_transaction(...) VALUES (...);

  ELSE
     ...remonter une erreur...
  END IF ;

COMMIT ;
```

Transfert entre portefeuilles

Deux problèmes principaux en cas d'exécutions concurrentes:

- Le BEGIN ne spécifie pas de niveau d’isolation, donc c'est *Read Committed* : en cas de modification concurrente, aucune erreur ne sera remontée.

- Si une autre transaction a changé le solde mais n’a pas encore committé, on teste un solde qui est déjà obsolète.

« Pluie ACIDe » ?

« ACIDRain: Concurrency-Related Attacks on Database-Backed Web Applications »

Par Todd Warszawski et Peter Bailis

http://www.bailis.org/papers/acidrain-sigmod2017.pdf

« In theory, database transactions protect application data from corruption and integrity violations. In practice, database transactions frequently execute under weak isolation that exposes programs to a range of concurrency anomalies, and programmers may fail to correctly employ transactions.»

« We apply a prototype 2AD analysis tool to 12 popular self-hosted eCommerce applications written in four languages and deployed on over 2M websites. We identify and verify 22 critical ACIDRain attacks that allow attackers to corrupt store inventory, over-spend gift cards, and steal inventory. »

Solutions ?

- ~~Mises à jour en trigger~~ (non)

- Verrouillage « manuel » préalable des lignes ou tables, pour implémenter une exclusion mutuelle, via:

  - SELECT FOR UPDATE ou UPDATE (par ligne)

  - LOCK TABLE (toute une table)

  - SELECT pg_advisory_xact_lock(…) (verrou « consultatif »)

- Ou Niveaux supérieurs d’isolation (Repeatable Read, **Serializable**)

Isolation : Read Committed

```
CREATE TABLE list (x int) ;
INSERT INTO list VALUES (1),(2),(3),(4) ;
```

Transaction Tx1

```
BEGIN ;

UPDATE list SET x=x-1 ;




COMMIT ;
```

Transaction Tx2

```
BEGIN ;



DELETE FROM list WHERE x
= (SELECT max(x) FROM
list) ;

COMMIT ;
```

Question : quel est le contenu de la table à l'issue de Tx1 et Tx2 ?

Isolation : Read Committed

Transaction Tx1

```
BEGIN ;

UPDATE list SET x=x-1 ;
UPDATE 4
```

Transaction Tx2

```
BEGIN ;

DELETE FROM list WHERE x =
(SELECT max(x) FROM list) ;
```

(attente jusqu'à fin de Tx1)

```
DELETE 0
COMMIT ;
```

Transaction Tx3

```
SELECT * FROM list ;
 x
---
 0
 1
 2
 3
```

Isolation : Repeatable Read

Transaction Tx1

```
BEGIN isolation level
repeatable read ;
```

```
UPDATE list SET x=x-1 ;
UPDATE 4
```

```
COMMIT ;
```

Transaction Tx2

```
BEGIN isolation level
repeatable read;
```

```
DELETE FROM list WHERE x =
(SELECT max(x) FROM list) ;
```

(attente jusqu'à fin de Tx1)

ERREUR: n'a pas pu sérialiser un accès à cause d'une mise à jour en parallèle

```
\echo :SQLSTATE
```

40001

Transaction Tx3

```
SELECT * FROM list ;
```

```
 x
---
 0
 1
 2
 3
```

Repeatable Read insuffisant?

```
CREATE TABLE list (id serial, couleur text);
INSERT INTO list(couleur) VALUES ('rouge'),('bleu'),('rouge'),('bleu'),
('rouge'),('bleu');
```

Transaction Tx0

```
BEGIN isolation
level repeatable
read ;

SELECT * FROM
list ORDER BY id;

 id | couleur
----+---------
  1 | rouge
  2 | bleu
  3 | rouge
  4 | bleu
  5 | rouge
  6 | bleu

COMMIT;
```

Transaction Tx1

```
BEGIN isolation level repeatable
read ;

UPDATE list SET couleur = 'rouge'
WHERE couleur <> 'rouge';

SELECT * FROM list ORDER BY id;

 id | couleur
----+---------
  1 | rouge
  2 | rouge
  3 | rouge
  4 | rouge
  5 | rouge
  6 | rouge

COMMIT;
```

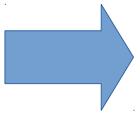
Transaction Tx2

```
BEGIN isolation level repeatable
read ;

UPDATE list SET couleur = 'bleu'
WHERE couleur <> 'bleu';

SELECT * FROM list ORDER BY id;

 id | couleur
----+---------
  1 | bleu
  2 | bleu
  3 | bleu
  4 | bleu
  5 | bleu
  6 | bleu

COMMIT;
```

Repeatable Read insuffisant?

État Initial

```
SELECT * FROM
list ORDER BY id;

 id | couleur
----+---------
  1 | rouge
  2 | bleu
  3 | rouge
  4 | bleu
  5 | rouge
  6 | bleu
```

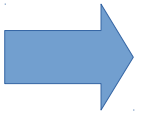


Transaction Tx1

en parallèle avec

Transaction Tx2

(pas d'erreur d'exécution)



État Final

```
SELECT * FROM
list ORDER BY id;

 id | couleur
----+---------
  1 | bleu
  2 | rouge
  3 | bleu
  4 | rouge
  5 | bleu
  6 | rouge
```

Problème: l'état final est différent des états produits par Tx1 suivi de Tx2 (tout en bleu), et de Tx2 suivi de Tx1 (tout en rouge).

Donc cette exécution concurrente présente une **anomalie de sérialisation**.

Isolation: Serializable

Transaction Tx1

```
=# BEGIN isolation level
serializable;
BEGIN
=# UPDATE list SET couleur = 'rouge'
WHERE couleur <> 'rouge';
UPDATE 3
=# SELECT * FROM list ORDER BY id;
 id | couleur
----+---------
  1 | rouge
  2 | rouge
  3 | rouge
  4 | rouge
  5 | rouge
  6 | rouge
=# COMMIT;
COMMIT
```

Transaction Tx2

```
=# BEGIN isolation level
serializable;
BEGIN
=# UPDATE list SET couleur = 'bleu'
WHERE couleur <> 'bleu';
UPDATE 3
=# SELECT * FROM list ORDER BY id;
 id | couleur
----+---------
  1 | bleu
  2 | bleu
  3 | bleu
  4 | bleu
  5 | bleu
  6 | bleu
```

```
=# COMMIT;
```

```
ERREUR:  n'a pas pu sérialiser un accès à cause des dépendances de lecture/écriture
parmi les transactions
DÉTAIL : Reason code: Canceled on identification as a pivot, during commit attempt.
ASTUCE : La transaction pourrait réussir après une nouvelle tentative.
```

Niveaux d’isolation

- `BEGIN TRANSACTION isolation level` **`Read Committed`** `;`

  Nouveau *snapshot* à chaque instruction, et même intra-instruction. On ne peut pas voir les changements faits par d’autres transactions avant leur COMMIT mais pour le reste, la concurrence est gérée en mode « advienne que pourra ».

- `BEGIN TRANSACTION isolation level` **`Repeatable Read`** `;`

  SI = *Snapshot Isolation*. Un seul snapshot pour toute la transaction. Les écritures concurrentes sont détectées et rejetées.

- `BEGIN TRANSACTION isolation level` **`Serializable`** `;`

  SSI = *Serializable Snapshot Isolation*. Aucune « anomalie » lié aux accès concurrents n’est tolérée. D'après la définition du standard: « *Toute exécution concurrente d'un jeu de transactions sérialisables doit apporter la garantie de produire le même effet que l'exécution consécutive de chacun d'entre eux dans un certain ordre* ».

On peut aussi choisir un niveau d’isolation au tout début de la transaction :

- `SET TRANSACTION isolation level {Read Committed , Repeatable Read, Serializable}` ; dès le début de la transaction

Et choisir un niveau d’isolation par défaut :

- `SET default_transaction_isolation TO {Read Committed , Repeatable Read, Serializable} ;`

Anomalies / Niveau d'isolation

https://docs.postgresql.fr/11/transaction-iso.html

Niveau d'isolation	Lecture sale	Lecture non reproductible	Lecture fantôme	Anomalie de sérialisation
Read Uncommited (en français, « Lecture de données non validées »)	Autorisé mais pas dans PostgreSQL	Possible	Possible	Possible
Read Commited (en français, « Lecture de données validées »)	Impossible	Possible	Possible	Possible
Repeatable Read (en français, « Lecture répétée »)	Impossible	Impossible	Autorisé mais pas dans PostgreSQL	Possible
Serializable (en français, « Sérialisable »)	Impossible	Impossible	Impossible	Impossible

Typologies illustrées d'anomalies de sérialisation:

https://wiki.postgresql.org/wiki/SSI

Serializable: avantages

On peut concevoir son enchaînement d'instructions SQL dans une transaction **comme si aucune autre transaction ne s'exécutait** en parallèle.

C'est le moteur SQL qui est chargé de **détecter** tout problème de concurrence dynamiquement, et d'**annuler** les transactions problématiques.

Serializable: inconvénients

- Il faut impérativement une stratégie de réessai des transactions qui échouent pour cause d'erreur de sérialisation.
- « Faux positifs » possibles (granularité des verrous de prédicats).
- Réduction possible des performances (voir les conseils de la documentation pour mitiger certains facteurs).
- Non disponible sur les instances en réplication.
- Avec Postgres ≤ 11, pas de parallélisme sur les index hash, GiST, GIN.
- Avec Postgres ≤ 12, pas de parallélisme intra-requête.

Conclusion

L'isolation « *Read Committed* » ne devrait être utilisée que quand elle est justifiée par des besoins spécifiques.

Ne soyez pas le prochain FlexCoin...