Concurrency Control in Multiuser Environments

credativ GmbH / credativ Ltd. / credativ, LLC

PostgreSQL Training

Introduction

What are the problems?

- Two reading transactions — no problem
- One reading transaction, one writing transaction — reading consistent data
- Two writing transactions — writing consistent data

We want

- Correct behaviour
- Good performance

`http://www.postgresql.org/docs/8.4/interactive/transaction-iso.html`

credativ

Problems

Dirty read A transaction reads data of a second transaction before the second transaction has committed.

Nonrepeatable read A transaction reads a row twice with different results.

Phantom read A transaction reads data restricted by a search condition. If the query is executed again with the same search condition, it returns a different result set.

Transaction Isolation Levels in Theory

Read uncommitted dirty read, nonrepeatable read, phantom read possible

Read committed no dirty read possible

Repeatable read no nonrepeatable read possible

Serializable no phantom read possible

credativ

In Practice

How does PostgreSQL process concurrent operations?

- Snapshots
- Conflict resolution

Snapshots

- Isolate concurrent transactions
- Each transaction sees a consistent "snapshot" of the database at a certain point of time
- Updating a row creates a new row version
- Old row version have to be garbage collected with `VACUUM`

Programming Rule

Avoid long running transactions
Snapshot data fills up the hard disk and wastes system resources

Transaction Isolation Levels in Practice

Read committed Snapshot is created at the beginning of each command.

Serializable Snapshot is created at the beginning of each transaction.

Read committed is the default isolation level.

Simple Cases

- Read-only transactions: trivial
- Read-write transaction: Snapshots protect reading transactions against writing transactions (prevent dirty reads).
  - Readers in serializable mode won't see any changes in the database after the beginning of their transaction.
  - Writers in read committed mode would see changes after each command.

Correct Reading Processes

```
BEGIN;
SELECT sum(balance) FROM accounts;
SELECT sum(branch_balance) FROM branches;
-- compare results
COMMIT;
```

Could be wrong in read committed mode

Programming Rule

Read-only processes might need serializable isolation level.

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
...
COMMIT;
```

PostgreSQL's MVCC Rule

Readers don't block writers; writers don't block readers.

Concurrent Writing

`UPDATE` or `DELETE` cannot write into the same row at the same time.
But they can write concurrently into different rows of the same table.

Concurrent Writing

What happens at concurrent writes into a row?

1. If the other transaction has not finished, wait for commit or rollback.
2. On rollback, continue
3. On commit:
   - In read committed mode: Read the row again, then continue
   - In serializable mode: Error

Concurrent Writing Example

You are counting hits on a web page:

```
UPDATE webpages SET hits = hits + 1 WHERE url = '...';
```

Is that safe with multiple concurrent accesses?

Possible Problem Without MVCC

```
UPDATE webpages SET hits = hits + 1 WHERE url = 'mypage';
```

Client 1	Client 2
reads hits = 531	
	reads hits = 531
hits + 1 = 532	
	hits + 1 = 532
writes hits = 532	
	writes hits = 532
commit	
	commit

hits is now 532, but correct value should be 533 (“lost update”).

No Problem in Read Committed Mode

```
UPDATE webpages SET hits = hits + 1 WHERE url = 'mypage';
```

Client 1	Client 2
reads hits = 531	
	reads hits = 531
hits + 1 = 532	
	hits + 1 = 532
writes hits = 532	
	tries to write, but has to wait
commit	
	re-reads row, sees hits = 532
	calculates hits + 1 = 533
	writes hits = 533
	commit

hits is now 533

Problem in Serializable Mode

```
UPDATE webpages SET hits = hits + 1 WHERE url = 'mypage';
```

Client 1	Client 2
reads hits = 531	
	reads hits = 531
hits + 1 = 532	
	hits + 1 = 532
writes hits = 532	
	tries to write, but has to wait
commit	
	serialization error

Client has to catch the error and retry the transaction.

Why Serializable Mode At All?

It works for more complex applications. Next example:

```
BEGIN;

SELECT hits FROM webpages WHERE url = 'mypage';

-- calculate $new = $hits + 1 internally

UPDATE webpages SET hits = $new WHERE url = 'mypage';

COMMIT;
```

Problem in Read Committed Mode

```
SELECT hits FROM webpages WHERE url = 'mypage';
-- calculate $new = $hits +1 internally
UPDATE webpages SET hits = $new WHERE url = 'mypage';
```

Client 1	Client 2
reads $hits = 531	
	reads $hits = 531
$new = 532	
	$new = 532
writes hits = 532	
	tries to write, but has to wait
commit	
	re-reads row, sees hits = 532
	calculates $new, still 532
	writes hits = 532; commit

Wrong result

Solution 1: SELECT FOR UPDATE

`SELECT FOR UPDATE` locks read rows.
Code example:

```
BEGIN;

SELECT hits FROM webpages WHERE url = 'mypage' FOR UPDATE;

-- calculate $new = $hits + 1 internally

UPDATE webpages SET hits = $new WHERE url = 'mypage';

COMMIT;
```

How SELECT FOR UPDATE Works

```
SELECT hits FROM webpages WHERE url = 'mypage' FOR UPDATE;
-- calculate $new = $hits + 1 internally
UPDATE webpages SET hits = $new WHERE url = 'mypage';
```

Client 1	Client 2
locks row, reads $hits = 531	
	tries to lock row, but has to wait
$new = 532, writes hits = 532	
commit	
	locks row, reads $hits = 532
	$new = 533, writes hits = 533
	commit

Solution 2: Serializable Mode

Pseudo-code example:

```
loop {
    BEGIN;
    SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

    SELECT hits FROM webpages WHERE url = 'mypage';
    -- calculate $new = $hits + 1 internally
    UPDATE webpages SET hits = $new WHERE url = 'mypage';

    if (no error)
        break loop;
    else
        ROLLBACK;
}
COMMIT;
```

Performance Comparison

- Read committed + `SELECT FOR UPDATE` is pessimistic locking.
- Serializable is optimistic locking.
- If chances for a conflict are high, then read committed wins, else serializable.

Update several rows

Money transfer of 100 euros from Alice to Bob in read committed mode:

```
BEGIN;

UPDATE account SET balance = balance - 100
    WHERE owner = 'Alice';

UPDATE account SET balance = balance + 100
    WHERE owner = 'Bob';

COMMIT;
```

That's correct, but ...

Deadlocks

Assume Bob transfers to Alice at the same time:

Client 1	Client 2
locks and updates row of Alice	
	locks and updates row of Bob
tries to lock row of Bob, but has to wait	
	tries to lock row of Alice, but has to wait

Nothing works anymore.

Resolving Deadlocks

- PostgreSQL detects the deadlock and cancels one of the transactions.
- A loop to retry the transaction is required.
- So it's not easier than serializable mode.

credativ

Avoiding Deadlocks

Possible methods to avoid deadlocks:

- Lock objects in the same order
- Set highest lock first, no upgrades

Often not realistic

Constraints Spanning Multiple Rows

We want to ensure that we always have at least 1000 euros total in our accounts:

```
BEGIN;

UPDATE my_accounts SET balance = balance - $withdrawal
    WHERE acctid = 'checking';

SELECT SUM(balance) AS sum FROM my_accounts;

if (sum >= 1000.00)
    COMMIT;
else
    ROLLBACK;
```

That is not safe, not even in serializable mode.

Problem Scenario

Assume we have 600 euros each in two accounts (= 1200 euros), and now we withdraw 200 euros from each at the same time (= 800 euros):

Client 1	Client 2
reads checking account, gets 600;	
changes checking balance to 400	
	reads savings account, gets 600;
	changes savings balance to 400
calculates sum:	
$400 + 600 = 1000$	
	calculates sum:
	$600 + 400 = 1000$
commit	
	commit

Observations

- Serializable mode isn't truly "serializable".
- PostgreSQL doesn't support predicate locking.

Solution: Explicit Locking

```
BEGIN;

LOCK TABLE my_accounts IN SHARE ROW EXCLUSIVE MODE;

UPDATE my_accounts SET balance = balance - $withdrawal
    WHERE acctid = 'checking';

SELECT SUM(balance) AS sum FROM my_accounts;

if (sum >= 1000.00)
    COMMIT;
else
    ROLLBACK;
```

Explicit Locking

Explicit lock modes:

```
LOCK TABLE name IN
     ACCESS SHARE MODE;
     ROW SHARE MODE;
     ROW EXCLUSIVE MODE;
     SHARE UPDATE EXCLUSIVE MODE;
     SHARE MODE;
     SHARE ROW EXCLUSIVE MODE;
     EXCLUSIVE MODE;
     ACCESS EXCLUSIVE MODE;
```

Most of them aren't needed by applications.

Explicit Locking

Lock without waiting:

```
LOCK TABLE name IN ... MODE NOWAIT;
```

Useful for special cases

Summary

Serializable mode:

- Snapshot at the beginning of a transaction
- Locks updated/deleted rows (error on conflict)
- Not truly serializable
- Tends to have less issues
- Clients should be prepared to retry transactions

Read committed mode:

- Snapshot at the beginning of a command
- Able to write updated/deleted rows (uses newest row version)
- Suitable for updates on single rows
- Client-side calculations force the use of `SELECT FOR UPDATE`

Explicit locking for constraints spanning multiple rows

credativ