SSI

From PostgreSQL wiki

Languages:	**English** • Français

Documentation of Serializable Snapshot Isolation (SSI) in PostgreSQL compared to plain Snapshot Isolation (SI). These correspond to the SERIALIZABLE and REPEATABLE READ transaction isolation levels, respectively, in PostgreSQL beginning with version 9.1.

Contents

Overview

With true serializable transactions, if you can show that your transaction will do the right thing if there are no concurrent transactions, it will do the right thing in any mix of serializable transactions or be rolled back with an error. The transaction can then be retried from the beginning. While the error is often a serialization failure, it can also be an error indicating a constraint violation. Any query results from a failed serializable transaction must be ignored; use the results from the retry.

This document shows problems which can occur with certain combinations of transactions at the REPEATABLE READ transaction isolation level, and how they are avoided at the SERIALIZABLE transaction isolation level beginning with PostgreSQL version 9.1.

This document is oriented toward the application programmer or database administrator. For internals of the SSI implementation, please see the Serializable Wiki page. For more information about how to use this isolation level, see the current PostgreSQL documentation (http://www.postgresql.org/docs/current/interactive/transaction-iso.html#XACT-SERIALIZABLE) .

Examples

In environments which avoid blocking-based integrity protection by counting on SSI, it will be common for the database to be configured (in postgresql.conf) with:

```
default_transaction_isolation = 'serializable'
```

Because of this, all examples were tested with this setting, and clutter is avoided by using a simple "begin" without explicitly declaring the transaction isolation level for each transaction.

Simple Write Skew

When two concurrent transactions each determine what they are writing based on reading a data set which overlaps what the other is writing, you can get a state which could not occur if either had run before the other. This is known as *write skew*, and is the simplest form of serialization anomaly against which SSI protects you.

When there is write skew in SSI, both transactions proceed until one transaction commits. The first committer wins and the other transaction is rolled back. The "first committer wins" rule ensures that there is progress and that the transaction which is rolled back can immediately be retried.

Black and White

In this case there are rows with a color column containing 'black' or 'white'. Two users concurrently try to make all rows contain matching color values, but their attempts go in opposite directions. One is trying to update all white rows to black and the other is trying to update all black rows to white.

If these updates are run serially, all colors will match. If they are run concurrently in REPEATABLE READ mode, the values will be switched, which is not consistent with any serial order of runs. If they are run concurrently in SERIALIZABLE mode, SSI will notice the write skew and roll back one of the transactions.

The example can be set up with these statements:

```
create table dots
 (
  id int not null primary key,
  color text not null
 );
insert into dots
```

```
with x(id) as (select generate_series(1,10))
select id, case when id % 2 = 1 then 'black'
  else 'white' end from x;
```

Black and White Example

session 1

```
begin;
update dots set color = 'black'
 where color = 'white';
```

session 2

```
begin;
update dots set color = 'white'
 where color = 'black';
```

At this point one transaction or the other is doomed to fail.

```
commit;
```

First commit wins.

```
select * from dots order by id;
```

```
 id | color
----+-------
  1 | white
  2 | white
  3 | white
  4 | white
  5 | white
  6 | white
  7 | white
  8 | white
  9 | white
 10 | white
(10 rows)
```

This one ran as if by itself.

```
commit;
```

```
ERROR:  could not serialize access
    due to read/write dependencies
    among transactions
DETAIL:  Cancelled on identification
```

```
      as a pivot, during commit attempt.
HINT:  The transaction might succeed if retried.
```

A serialization failure. We roll back and try again.

```
rollback;
begin;
update dots set color = 'black'
  where color = 'white';
commit;
```

No concurrent transaction to interfere.

```
select * from dots order by id;
```

```
 id | color
----+-------
  1 | black
  2 | black
  3 | black
  4 | black
  5 | black
  6 | black
  7 | black
  8 | black
  9 | black
 10 | black
(10 rows)
```

This transaction ran by itself, after the other.

Intersecting Data

This example is taken from the PostgreSQL documentation. Two concurrent transactions read data, and each uses it to update the range read by the other. A simple, though somewhat contrived, example of data skew.

The example can be set up with these statements:

```
CREATE TABLE mytab
(
  class int NOT NULL,
  value int NOT NULL
);
INSERT INTO mytab VALUES
(1, 10), (1, 20), (2, 100), (2, 200);
```

Intersecting Data Example

session 1

```
BEGIN;
SELECT SUM(value) FROM mytab WHERE class = 1;
```

```
 sum
-----
  30
(1 row)
```

```
INSERT INTO mytab VALUES (2, 30);
```

session 2

```
BEGIN;
SELECT SUM(value) FROM mytab WHERE class = 2;
```

```
 sum
-----
 300
(1 row)
```

```
INSERT INTO mytab VALUES (1, 300);
```

Each transaction has modified what the other transaction would have read. If both were allowed to commit, this would break serializable behavior, because if they were run one at a time, one of the transactions would have seen the INSERT the other committed. We wait for a successful COMMIT of one of the transactions before we roll anything back, though, to ensure progress and prevent thrashing.

```
COMMIT;
```

```
COMMIT;
```

```
ERROR:  could not serialize access
      due to read/write dependencies
      among transactions
DETAIL:  Cancelled on identification
       as a pivot, during commit attempt.
HINT:  The transaction might succeed if retried.
```

So now we roll back the failed transaction and retry it from the beginning.

```
ROLLBACK;
BEGIN;
SELECT SUM(value) FROM mytab WHERE class = 1;
```

```
 sum
-----
 330
(1 row)
```

```
INSERT INTO mytab VALUES (2, 330);
COMMIT;
```

This succeeds, leaving an end result consistent with a serial execution of the transactions.

```
SELECT * FROM mytab;
```

```
 class | value
-------+-------
     1 |    10
     1 |    20
     2 |   100
     2 |   200
     1 |   300
     2 |   330
(6 rows)
```

Overdraft Protection

The hypothetical case is that there is a bank which allows depositors to withdraw money up to the total of what they have in all accounts. The bank will later automatically transfer funds as needed to close the day with a positive balance in each account. Within a single transaction they check that the total of all accounts exceeds the amount requested.

Someone's trying to get clever and trick the bank by submitting $900 withdrawals to two accounts with $500 balances simultaneously. At the REPEATABLE READ transaction isolation level, that could work; but if the SERIALIZABLE transaction isolation level is used, SSI will detect a "dangerous structure" in the read/write pattern and reject one of the transactions.

The example can be set up with these statements:

```
create table account
 (
  name text not null,
  type text not null,
  balance money not null default '0.00'::money,
  primary key (name, type)
 );
insert into account values
 ('kevin','saving', 500),
 ('kevin','checking', 500);
```

Overdraft Protection Example

session 1

```
begin;
select type, balance from account
 where name = 'kevin';
```

```
  type   | balance
---------+---------
saving   | $500.00
checking | $500.00
(2 rows)
```

The total is $1000, so a $900 withdrawal is OK.

session 2

```
begin;
select type, balance from account
 where name = 'kevin';
```

```
  type   | balance
---------+---------
saving   | $500.00
checking | $500.00
(2 rows)
```

The total is $1000, so a $900 withdrawal is OK.

```
update account
 set balance = balance - 900::money
 where name = 'kevin' and type = 'saving';
```

So far everything's OK.

```
update account
 set balance = balance - 900::money
 where name = 'kevin' and type = 'checking';
```

Now we have a problem. This can't co-exist with the other transaction's activity. We don't cancel yet, because the transaction would fail on the same conflicts if retried. The first committer will win, and the other transaction will fail when it tries to continue after that.

```
commit;
```

This one happened to commit first. Its work is persisted.

```
commit;
```

```
ERROR:  could not serialize access
      due to read/write dependencies
      among transactions
DETAIL:  Cancelled on identification
       as a pivot, during commit attempt.
HINT:  The transaction might succeed if retried.
```

This transaction failed to withdraw the money. Now we roll back and retry the transaction.

```
rollback;
begin;
select type, balance from account
 where name = 'kevin';
```

```
  type   | balance
---------+----------
saving   | -$400.00
checking |  $500.00
(2 rows)
```

We see they have a net of $100. This request for $900 will be rejected by the application.

Three or More Transactions

Serialization anomalies can result from more complex patterns of access, involving three or more transactions.

Primary Colors

This is similar to the "Black and White" write skew example, except that we're using the three primary colors. One transaction is trying to update red to yellow, the next is trying to update yellow to blue, and the third is trying to update blue to red. If these were executed one at a time, you would be left with either one or two colors in the table, depending on the order of execution. If any two are executed concurrently, the one trying to read the rows being updated by the other will appear to execute first, since it won't see the work of the other transaction, so there is no problem there. Whether the other transaction is run before that or after that, the results are consistent with some serial order of execution.

If all three are run concurrently, there is a cycle in the apparent order of execution. A Repeatable Read transaction would not detect this, and the table would still have three colors. A Serializable transaction will detect the problem and roll one of the transactions back with a serialization failure.

The example can be set up with these statements:

```
create table dots
 (
  id int not null primary key,
  color text not null
 );
insert into dots
 with x(id) as (select generate_series(1,9000))
 select id, case when id % 3 = 1 then 'red'
  when id % 3 = 2 then 'yellow'
  else 'blue' end from x;
create index dots_color on dots (color);
analyze dots;
```

Primary Colors Example

session 1	session 2	session 3

`begin;`
`update dots set color = 'yellow'`
`where color = 'red';`		

	`begin;`
	`update dots set color = 'blue'`
	`where color = 'yellow';`	

		`begin;`
		`update dots set color = 'red'`
		`where color = 'blue';`

At this point at least one of these three transactions is doomed to fail. To ensure progress, we wait until one of them commits. The commit will succeed, which will not only ensure that progress is made but that an immediate retry of a failed transaction won't fail again *on the same combination of transactions*.

```
commit;
```

First commit wins. Session 2 is bound to fail at this point, because during commit it was determined that it had the better chance of succeeding if retried immediately.

```
select color, count(*) from dots
  group by color
  order by color;
```

```
color | count
-------+-------
blue  | 3000
yellow | 6000
(2 rows)
```

This appears to have run before the other updates.

```
commit;
```

This works if attempted at this point. If session 2 does more work first, this transaction might also need to be cancelled and retried.

```
select color, count(*) from dots
 group by color
 order by color;
```

```
color  | count
--------+-------
red    | 3000
yellow |  6000
(2 rows)
```

This appears to have run after the transaction on session 1.

```
commit;
```

```
ERROR:  could not serialize access
      due to read/write dependencies
      among transactions
DETAIL:  Cancelled on identification
       as a pivot, during commit attempt.
HINT:  The transaction might succeed if retried.
```

A serialization failure. We roll back and try again.

```
rollback;
begin;
update dots set color = 'blue'
 where color = 'yellow';
commit;
```

Things are OK on retry.

```
select color, count(*) from dots
 group by color
 order by color;
```

```
 color | count
-------+-------
 blue  |  6000
 red   |  3000
(2 rows)
```

This appears to have run last, which it did.

An interesting point is that if session 2 attempted to commit after session 1 and before session 3, it would still have failed, and a retry would still have succeeded, but the fate of the transaction on session 3 is not deterministic. It might have succeeded or it might have gotten a serialization failure and required a retry.

This is because the predicate locking used as part of conflict detection works based on pages and tuples actually accessed, and there is a random factor used in inserting index entries which have equal keys, in order to minimize contention; so even with identical run sequences it is possible to see differences in where serialization failures occur. That is why it is important, when relying on serializable transactions for managing concurrency, to have some generalized technique for identifying serialization failures and retrying transactions from the beginning.

It is also worth noting that if session 2 committed the retry transaction before session 3 committed its transaction, any subsequent query which viewed rows successfully updated from yellow to blue by session 2 would deterministically doom the transaction on session 3, because these would not be rows which session 3 would see as blue and update to red. For the transaction on session 3 to successfully commit, it must be considered to have run before the successful transaction on session 2, so exposing a state in which the work of the transaction on session 2 is visible, but not the work of the transaction on session 3, means that the transaction on session 3 must fail. The act of *observing* a recently modified database state can cause serialization failures. This will be further explored in other examples.

Enforcing Business Rules in Triggers

If all transactions are serializable, business rules can be enforced in triggers without the problems associated with other transaction isolation levels. Where a declarative constraint works, it will generally be faster, easier to implement and maintain, and less prone to bugs -- so triggers should only be used this way where a declarative constraint won't work.

Unique-Like Constraints

Say you want something similar to a unique constraint, but it's a little more complicated. For this example, we want uniqueness in the first six characters of the text column.

The example can be set up with these statements:

```
create table t (id int not null, val text not null);
with x (n) as (select generate_series(1,10000))
  insert into t select x.n, md5(x.n::text) from x;
alter table t add primary key(id);
create index t_val on t (val);
vacuum analyze t;
create function t_func()
  returns trigger
  language plpgsql as $$
declare
  st text;
```

```
begin
 st := substring(new.val from 1 for 6);
 if tg_op = 'UPDATE' and substring(old.val from 1 for 6) = st then
  return new;
 end if;
 if exists (select * from t where val between st and st || 'z') then
  raise exception 't.val not unique on first six characters: "%"', st;
 end if;
 return new;
end;
$$;
create trigger t_trig
 before insert or update on t
 for each row execute procedure t_func();
```

To confirm that the trigger is enforcing the business rule when there are no concurrency issues, on a single connection:

```
insert into t values (-1, 'this old dog');
insert into t values (-2, 'this old cat');
```

```
ERROR:  t.val not unique on first six characters: "this o"
```

Now we try in two concurrent sessions.

Unique-Like Constraint Example

session 1	**session 2**
`begin;` `insert into t values (-3, 'the river flows');`	
	`begin;` `insert into t values (-4, 'the right stuff');`
	This works for the moment, because the work of the other transaction is not visible to this transaction, but both transactions may not commit without violating the business rule.
	`commit;`
	The first committer wins. This transaction is safe.
A commit here would fail, but so would an attempt to run any other statement within this doomed transaction.	

```
select * from t where id < 0;
```

```
ERROR:  could not serialize access
      due to read/write dependencies
      among transactions
DETAIL:  Canceled on identification as a pivot,
       during conflict out checking.
HINT:  The transaction might succeed if retried.
```

Since this is a serialization failure, the transaction should be retried.

```
rollback;
begin;
insert into t values (-3, 'the river flows');
```

On retry we get an error which is more helpful to the user.

```
ERROR:  t.val not unique on first six characters: "the ri"
```

FK-Like Constraints

Sometimes two tables must have a relationship very similar to a foreign key relationship, but there are extra criteria which makes a foreign key insufficient to completely cover the necessary integrity checking. In this example a project table contains a reference to a person table's key in a project_manager column, but not just *any* person will do; the person specified must be flagged as a project manager.

The example can be set up with these statements:

```
create table person
 (
  person_id int not null primary key,
  person_name text not null,
  is_project_manager boolean not null
 );
create table project
 (
  project_id int not null primary key,
  project_name text not null,
  project_manager int not null
 );
create index project_manager
 on project (project_manager);

create function person_func()
```

```
 returns trigger
 language plpgsql as $$
begin
 if tg_op = 'DELETE' and old.is_project_manager then
  if exists (select * from project
         where project_manager = old.person_id) then
   raise exception
    'person cannot be deleted while manager of any project';
  end if;
 end if;
 if tg_op = 'UPDATE' then
  if new.person_id is distinct from old.person_id then
   raise exception 'change to person_id is not allowed';
  end if;
  if old.is_project_manager and not new.is_project_manager then
   if exists (select * from project
           where project_manager = old.person_id) then
     raise exception
      'person must remain a project manager while managing any projects';
   end if;
  end if;
 end if;
 if tg_op = 'DELETE' then
  return old;
 else
  return new;
 end if;
end;
$$;
create trigger person_trig
 before update or delete on person
 for each row execute procedure person_func();

create function project_func()
 returns trigger
 language plpgsql as $$
begin
 if tg_op = 'INSERT'
 or (tg_op = 'UPDATE' and new.project_manager <> old.project_manager) then
  if not exists (select * from person
              where person_id = new.project_manager
                and is_project_manager) then
   raise exception
    'project_manager must be defined as a project manager in the person table';
  end if;
 end if;
 return new;
end;
$$;
create trigger project_trig
 before insert or update on project
 for each row execute procedure project_func();

insert into person values (1, 'Kevin Grittner', true);
```

```
insert into person values (2, 'Peter Parker', true);
insert into project values (101, 'parallel processing', 1);
```

FK-Like Constraints Example

session 1

One person is being updated to no longer be a project manager.

```
begin;
update person
 set is_project_manager = false
 where person_id = 2;
```

session 2

At the same time, a project is being updated to make that person the manager of that project.

```
begin;
update project
 set project_manager = 2
 where project_id = 101;
```

These can't both be committed. The first commit will win.

```
commit;
```

The assignment of the person to the project commits first, so the other transaction is now doomed to fail. If either transaction had run at a different isolation level, both transactions could have committed, resulting in a violation of the business rules.

```
commit;
```

```
ERROR:  could not serialize access
    due to read/write dependencies
    among transactions
DETAIL:  Cancelled on identification
     as a pivot, during commit attempt.
HINT:  The transaction might succeed if retried.
```

A serialization failure. We roll back and try again.

```
rollback;
begin;
update person
  set is_project_manager = false
  where person_id = 2;
```

```
ERROR:  person must remain a project manager
      while managing any projects
```

On the retry we get a meaningful message.

Read Only Transactions

While a Read Only transaction cannot contribute to an anomaly which persists in the database, under Repeatable Read transaction isolation it can *see* a state which is not consistent with any serial (one-at-a-time) execution of transactions. A Serializable transaction implemented with SSI will never see such transient anomalies.

Deposit Report

A general class of problems involving read only transactions is batch processing, where one table controls which batch is currently the target of inserts. A batch is closed by updating the control table, at which point the batch is considered "locked" against further change, and processing of that batch occurs.

A particular example of this which occurs in real-world bookkeeping is receipting. Receipts might be added to a batch identified by the deposit date, or (if more than one deposit per day is possible) an abstract receipt batch number. At some point during the day, while the bank is still open, the batch is closed, a report is printed of the money received, and the money is taken to the bank for deposit.

The example can be set up with these statements:

```
create table control
  (
    deposit_no int not null
  );
insert into control values (1);
create table receipt
  (
    receipt_no serial primary key,
    deposit_no int not null,
    payee text not null,
    amount money not null
  );
insert into receipt
```

```
 (deposit_no, payee, amount)
 values ((select deposit_no from control), 'Crosby', '100');
insert into receipt
 (deposit_no, payee, amount)
 values ((select deposit_no from control), 'Stills', '200');
insert into receipt
 (deposit_no, payee, amount)
 values ((select deposit_no from control), 'Nash', '300');
```

Deposit Report Example

session 1 **session 2**

At a receipting counter, another receipt is added to the current batch.

```
begin;  -- T1
insert into receipt
 (deposit_no, payee, amount)
 values
 (
  (select deposit_no from control),
  'Young', '100'
 );
```

This transaction can see its own insert, but it's not visible to other transactions until commit.

```
select * from receipt;
```

```
 receipt_no | deposit_no | payee  | amount
------------+------------+--------+---------
          1 |          1 | Crosby | $100.00
          2 |          1 | Stills | $200.00
          3 |          1 | Nash   | $300.00
          4 |          1 | Young  | $100.00
(4 rows)
```

At about the same time, a supervisor clicks a button to close the receipt batch.

```
begin;  -- T2
select deposit_no from control;
```

```
 deposit_no
------------
          1
(1 row)
```

The application notes the receipting batch which is about to be closed, increments the batch number, and saves that to the control table.

```
update control set deposit_no = 2;
commit;
```

T1, the transaction inserting the last receipt for the old batch, hasn't committed yet even though the batch has been closed. If T1 commits before anyone looks at the contents of the closed batch, everything is OK. So far we don't have a problem; the receipt *appears* to have added before the batch was closed. We have behavior which is consistent with some one-at-a-time execution of the transactions: T1 -> T2.

For purposes of demonstration, we'll have the deposit report start before that last receipt commits.

```
begin;  -- T3
select * from receipt where deposit_no = 1;
```

```
 receipt_no | deposit_no | payee  | amount
------------+------------+--------+---------
          1 |          1 | Crosby | $100.00
          2 |          1 | Stills | $200.00
          3 |          1 | Nash   | $300.00
(3 rows)
```

Now we have a problem. T3 was started with the knowledge that T2 committed successfully, so T3 must be considered to have run after T2. (This could also be true if the T3 had run independently and selected from the control table, seeing the updated deposit_no.) But T3 cannot see the work of T1, so T1 appears to have run after T3. So we have a cycle T1 -> T2 -> T3 -> T1. And this would be a problem in practical terms; the batch is supposed to be closed and immutable, but a change will pop up late -- perhaps after the trip to the bank.

Under the REPEATABLE READ isolation level this would silently proceed without the anomaly being noticed. Under the SERIALIZABLE isolation level one of the transactions will be rolled back to protect the integrity of the system. Since a rollback and retry of T3 would hit the same error if T1 was still active, PostgreSQL will cancel T1, so that an immediate retry will succeed.

```
commit;
```

```
ERROR:  could not serialize access
      due to read/write dependencies
      among transactions
DETAIL:  Cancelled on identification
       as a pivot, during commit attempt.
HINT:  The transaction might succeed if retried.
```

OK, let's retry.

```
rollback;
begin;  -- T1 retry
insert into receipt
 (deposit_no, payee, amount)
 values
 (
  (select deposit_no from control),
  'Young', '100'
 );
```

What does the receipt table look like now?

```
select * from receipt;
```

```
 receipt_no | deposit_no | payee  | amount
------------+------------+--------+---------
          1 |          1 | Crosby | $100.00
          2 |          1 | Stills | $200.00
          3 |          1 | Nash   | $300.00
          5 |          2 | Young  | $100.00
(4 rows)
```

The receipt now falls into the next batch, making the deposit report in T3 correct!

```
commit;
```

No problem now.

```
commit;
```

This would have been OK anytime after T3's SELECT.

Rollover

While the read only transaction itself is rarely rolled back to prevent serialization failures, it will happen if the other transactions have already committed.

The example can be set up with these statements:

```
create table rollover (id int primary key, n int not null);
insert into rollover values (1,100), (2,10);
```

Rollover Example

session 1	session 2
One transaction looks at row 2 and updates row 1.	

`begin;  -- T1`
`update rollover`
`  set n = n + (select n from rollover where id = 2)`
`  where id = 1;`	

	At about the same time, another connection commits a change to row 2.

	`begin;  -- T2`
	`update rollover set n = n + 1 where id = 2;`
	`commit;`

	There is no problem so far; T1 appears to have executed first, since it saw row 2 without the change committed by T2.
	Now another transaction starts and needs to acquire a snapshot.
	`begin transaction read only;  -- T3` `select count(*) from pg_class;`
	T3 is running with a snapshot which can see the work of T2 but not T1, even though we've already established that T1 appears to have executed before T2. There is no problem, though, as long as T3 doesn't look at data modified by T1.

T1 can commit without error.

```
commit;
```

Now if the read only transaction attempts to read data modified by T1 there will be a cycle in the apparent order of execution, so the attempt must be rolled back.

```
select n from rollover where id in (1,2);
```

```
ERROR:  could not serialize access
      due to read/write dependencies
      among transactions
DETAIL:  Reason code: Canceled on conflict out
       to pivot 1117, during read.
HINT:  The transaction might succeed if retried.
```

We retry the transaction.

```
rollback;
begin transaction read only;  -- T3 retry
select count(*) from pg_class;
select n from rollover where id in (1,2);
commit;
```

```
 n
-----
 110
  11
(2 rows)
```

Now we see a view of the data consistent with T1 having run before T2.

Retrieved from "https://wiki.postgresql.org/index.php?title=SSI&oldid=34576"
Category: Documentation

- This page was last modified on 22 January 2020, at 03:09.