

# Transaction isolation levels and anomalies in PostgreSQL

# Agenda

- The “mandatory” boring slides
  - SQL92 Isolation levels
  - Isolation levels phenomena
- The Lost update issue
- Read Committed
- Examples
- Optional trip to other RDBMS

# PostgreSQL 14 adoption

- Who is already using PostgreSQL 14 ?

# PostgreSQL 14 - BUG #17485

- BUG #17485: Records missing from Primary Key index when doing REINDEX INDEX CONCURRENTLY
  - not only PK index, actually any index
- PostgreSQL 13.7 - no issues
- Check [email thread](#) and wait for a fix if using `concurrently` option
  - Fix release *probably* in August 11th release, unless a maintenance release out of regular schedule

# Reading (spoilers)

- [PostgreSQL manuals](#)
- [The Internals of PostgreSQL : Chapter 5 Concurrency Control](#)
- [MVCC in PostgreSQL – 1. Isolation](#)
- [PostgreSQL Concurrency Issues](#)

# Documentation

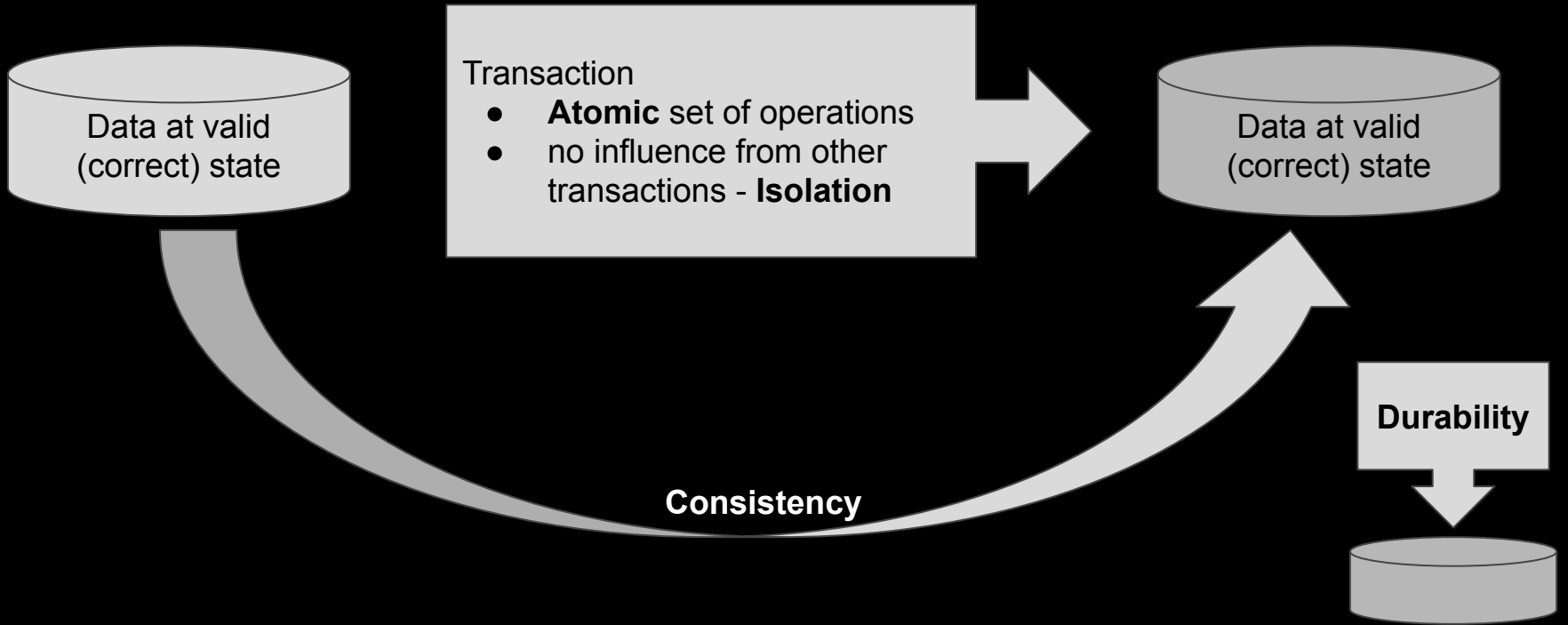
SQL standard SQL 1992, chapter 4.28 SQL-transactions and subsequent versions.

Exhaustive information about Postgres implementation can be found in the Postgres manuals.

ACID, ACID, ...



# transaction



# ANSI SQL

SQL-transaction has an isolation levels

- READ UNCOMMITTED
- READ COMMITTED
- REPEATABLE READ
- SERIALIZABLE

“The isolation level of a SQL-transaction is SERIALIZABLE by default. (by the SQL standard, *not default isolation level in most present databases available....*)

A serializable execution is defined to be an execution of the operations of concurrently executing SQL-transactions that produces the same effect as some serial execution of those same SQL-transactions.”

The four isolation levels guarantee that each SQL-transaction will be executed completely or not at all, and that **no updates will be lost**.



## Concurrent SQL-transactions phenomena (anomalies):

- Dirty read (read uncommitted)
- Non-repeatable read
  - T1 reads a row
  - T2 modified or deletes the same row and **commits**
  - T1 re-reads the row and see modified ones (or realize that the row no longer exists)
- Phantom read
  - T1 reads a set of rows satisfying some predicate
  - T2 create row(s) that satisfying predicate used by T1 **and commits**
  - T1 repeats the same reads using the same predicate, it receives different set of data
- Serialization anomaly
  - The result of successfully committing a group of transactions is inconsistent with all possible orderings of running those transactions one at a time.

# ANSI SQL92

Anomalies and transaction isolation levels:

Trn. Isolation level	Dirty read	Non-repeatable read	Phantom read
READ UNCOMMITTED	✓	✓	✓
READ COMMITTED	✗	✓	✓
REPEATABLE READ	✗	✗	✓
SERIALIZABLE	✗	✗	✗

# anomalies

Anomalies happens when multiple transactions, all of them itself correct, running together works incorrectly.

**Reading uncommitted changes** (dirty reads) is an **example** as transaction producing the changes might be rolled back, but other transactions will use the intermittent data state and can therefore produce **incorrect results**.

Serializable transaction isolation level is safe, but at the most probably not acceptable impact on performance (throughput).

# Transaction isolation levels in Postgres

Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read	Serialization Anomaly
Read uncommitted	✓, in PG ✗	✓	✓	✓
Read committed	✗	✓	✓	✓
Repeatable read	✗	✗	✓, in PG ✗	✓
Serializable	✗	✗	✗	✗

serialization anomaly: The result of successfully committing a group of transactions is inconsistent with all possible orderings of running those transactions one at a time.

# MVCC snapshots

PostgreSQL data consistency is maintained by using a multiversion model (Multiversion Concurrency Control, MVCC).

**This means that each SQL statement sees a snapshot of data** (a database version) as it was some time ago, regardless of the current state of the underlying data.

... providing **transaction isolation** for each database session

A tuple is a row version with **xmin** and **xmax** transaction identifiers (there are more attributes).

# lost update

SQL standard prohibit lost update as mentioned before (SQL92 - 4.28).  
Postgres works that way - updating a row requires level lock.  
It does not prevent **application logic to lost an update**.

# lost update

SQL standard prohibit lost update as mentioned before (SQL92 - 4.28).

Postgres works that way - updating a row requires level lock.

It does not prevent **application logic to lost an update.**

```
START TRANSACTION;
```

```
SELECT amount as v_amount FROM stock_item  
WHERE stock_item_id = 1  
\gset
```

```
\echo :v_amount  
10
```

```
SELECT amount as v_amount  
FROM stock_item  
WHERE stock_item_id = 1 \gset
```

```
UPDATE stock_item  
SET amount = :v_amount + 5 WHERE ...;
```

```
table stock_item;  
stock_item_id | item_id | amount  
-----+-----+-----  
1 | 1 | 15
```

# lost update...

Do not expect 16 in amount column. That is an application stuff in our example.

Takeaway: do not store values to be updated in application variables...

```
UPDATE stock_item SET amount = :v_amount + 1 WHERE ...;  
commit;
```

```
table stock_item;  
  stock_item_id | item_id | amount  
-----+-----+-----  
              1 |        1 |      11
```



# no lost update on a locked row

```
table stock_item;  
  stock_item_id | item_id | amount  
-----+-----+-----  
                1 |        1 |      11
```

```
START TRANSACTION;  
UPDATE stock_item SET amount = amount + 4 WHERE ...;
```

```
table stock_item;  
  stock_item_id | item_id | amount  
-----+-----+-----  
                1 |        1 |      15
```

← UPDATE stock\_item  
SET amount = amount + 5  
WHERE ...;

waiting...

# no lost update on a locked row

```
COMMIT;
```

```
table stock_item;  
  stock_item_id | item_id | amount  
-----+-----+-----  
                1 |        1 |      20
```

```
UPDATE stock_item  
SET amount = amount + 5;
```

waiting...

```
re-read the data before  
updating them =>  
no data loss
```

# No lost update...

Takeaway: do not store values to be updated in application variables... unless you have to (complex application logic), then...

Use **SELECT ... FOR UPDATE** - it'll place a lock on the row to be updated, wait if needed so it returns valid (latest) row content - and therefore data can differ from a simple select in read committed isolation level.

An Update after waiting for a lock also **re-check predicates** if no longer satisfied, row is not updated.

# Read committed

- No dirty reads in PostgreSQL.
- Non-repeatable reads.
- Inconsistent reads (result of non-repeatable reads).

# Read committed & Nonrepeatable Read

Obviously same as read uncommitted as stated in the manuals.

```
start transaction;

table stock_item;
  stock_item_id | item_id | amount
-----+-----+-----
              1 |         1 |         10
(1 row)
```

in autocommit (the default)

← UPDATE stock\_item SET amount = 20  
WHERE stock\_item\_id = 1;

```
table stock_item;
  stock_item_id | item_id | amount
-----+-----+-----
              1 |         1 |         20
(1 row)
```

# Read committed - antipatterns

```
table stock_item;  
  stock_item_id | item_id | amount  
-----+-----+-----  
                1 |        1 |      10
```

```
DO LANGUAGE plpgsql $code$  
BEGIN  
  -- do not allow negative amount for an item in stock  
  IF (SELECT amount FROM stock_item, pg_sleep(10) WHERE stock_item_id = 1) >= 10  
  THEN  
    UPDATE stock_item SET amount = amount - 10 WHERE stock_item_id = 1;  
  END IF;  
END;  
$code$;  
-- waiting for the pg_sleep()
```

```
table stock_item;  
  stock_item_id | item_id | amount  
-----+-----+-----  
                1 |        1 |     -2
```

```
UPDATE stock_item SET amount = amount - 2  
WHERE stock_item_id = 1;
```

```
  stock_item_id | item_id | amount  
-----+-----+-----  
                1 |        1 |      8
```

# Read committed - use constraints

```
table stock_item;  
 stock_item_id | item_id | amount  
-----+-----+-----  
              1 |        1 |      10
```

```
DO LANGUAGE plpgsql $code$  
BEGIN  
  -- do not allow negative amount for an item in stock  
  IF (SELECT amount FROM stock_item pg_sleep(10) WHERE stock_item_id = 1) >= 10 THEN  
    UPDATE stock_item SET amount = amount - 10 WHERE stock_item_id = 1;  
  END IF;  
END;
```

```
$code$;
```

```
--- waiting for the sleep
```

```
← UPDATE stock_item SET amount = amount - 2 WHERE stock_item_id = 1;
```

```
ERROR: new row for relation "stock item" violates check constraint "negative_amount_chk"
```

```
DETAIL: Failing row contains (1, 1, -2).
```

```
CONTEXT: SQL statement "UPDATE stock_item SET amount = amount - 10 WHERE stock_item_id = 1"
```

```
PL/pgSQL function inline_code_block line 5 at SQL statement
```

```
table stock_item;  
 stock_item_id | item_id | amount  
-----+-----+-----  
              1 |        1 |        8
```

# Read Committed

*Read Committed* is the default isolation level in PostgreSQL.

When a transaction uses this isolation level, a **SELECT** query (without a FOR UPDATE/SHARE clause) **sees only data committed before the query began; it never sees either uncommitted data or changes committed during query execution by concurrent transactions.**

In effect, a **SELECT** query **sees a snapshot of the database as of the instant the query begins to run.**

However, SELECT does see the effects of previous updates executed within its own transaction, even though they are not yet committed. Also note that two successive SELECT commands can see different data, even though they are within a single transaction, if other transactions commit changes after the first SELECT starts and before the second SELECT starts.



# Examples

## Concurrent transactions mixing

Read Committed and Repeatable read isolation level transactions in PostgreSQL.

- Read Write -Transactions usually in Read Committed
  - waits for a lock, if needed
- Read Only - short transactions or some ETL processes might benefit from Repeatable Read isolation level
  - in case of read-write transactions be ready to handle “serialization error” at application level (i.e. retry the failed operation)

# PostgreSQL

default:  
autocommit

```
tril=# CREATE TABLE foo(id int, val int);
```

```
tril=# INSERT INTO foo VALUES (1, 1);
```

```
tril=# START TRANSACTION ISOLATION LEVEL REPEATABLE  
READ;
```

```
tril=*# SELECT * FROM foo;  
 id | val  
----+-----  
(0 rows)
```

```
tril=*# SELECT * FROM foo;
```

???

# PostgreSQL

default:  
autocommit

```
tril=# CREATE TABLE foo(id int, val int);
```

```
tril=# INSERT INTO foo VALUES (1, 1);
```

```
tril=# START TRANSACTION ISOLATION LEVEL REPEATABLE  
READ;
```

```
tril=*# SELECT * FROM foo;  
 id | val  
----+-----  
(0 rows)
```

```
tril=*# SELECT * FROM foo;  
 id | val  
----+-----  
(0 rows)
```

```
tril=*# COMMIT;
```

```
tril=# SELECT * FROM foo;  
 id | val  
----+-----  
  1 |   1  
(1 row)
```

# PostgreSQL

default:  
autocommit

```
tril=# CREATE TABLE foo(id int, val int);  
tril=# INSERT INTO foo VALUES (1, 1);
```

```
tril=# CREATE TABLE bar(id int, val int);  
tril=# INSERT INTO bar VALUES (1, 1);
```

```
tril=# START TRANSACTION ISOLATION LEVEL REPEATABLE  
READ;
```

```
tril=*# SELECT * FROM foo;  
 id | val  
----+-----  
  1 |   1
```

```
tril=*# SELECT * from bar;
```

???

# PostgreSQL

default:  
autocommit

```
tril=# CREATE TABLE foo(id int, val int);  
tril=# INSERT INTO foo VALUES (1, 1);
```

```
tril=# CREATE TABLE bar(id int, val int);  
tril=# INSERT INTO bar VALUES (1, 1);
```

```
tril=# START TRANSACTION ISOLATION LEVEL REPEATABLE  
READ;
```

```
tril=*# SELECT * FROM foo;  
 id | val  
----+-----  
  1 |   1
```

```
tril=*# SELECT * from bar;  
 id | val  
----+-----  
(0 rows)
```

# PostgreSQL

```
tril=# START TRANSACTION;  
tril=# CREATE TABLE foo(id int, val int);
```

DDL is transactional

```
tril=# SELECT * FROM foo;  
ERROR: relation "foo" does not exist  
LINE 1: SELECT * FROM foo;
```

- DDL is transactional
  - [see the difference with MySQL example](#) where DDL commits active transaction
- Not only [PLPGSQL but also SQL functions](#) has volatility category as demonstrated in a 2020 p2d2.cz slides (function inlining).

# MVCC based snapshots

- **Read committed**

*SELECT* query sees a snapshot of the database as of the instant the query begins to run.

- Subsequent queries obtains their own snapshot

- **Repeatable read** (and Serializable)

snapshot is taken *at the first query after transaction start*, not at the `START TRANSACTION` statement itself.

- Thanks to MVCC writers do not block reader and reader does not block writers processes (row versioning - tuple [xmin, xmax])

# Triggers...

```
CREATE OR REPLACE FUNCTION trg_slow() RETURNS
TRIGGER LANGUAGE PLPGSQL AS
$func$
DECLARE
    v_seconds INT;
BEGIN

    IF TG_NARGS = 1 THEN
        v_seconds:= TG_ARGV[0]::int;
    ELSE
        v_seconds:= 0;
    END IF;

    perform pg_sleep(v_seconds);

    IF (TG_OP = 'DELETE') THEN
        RETURN OLD;
    ELSE
        RETURN NEW;
    END IF;

END;
$func$;
```

```
CREATE OR REPLACE FUNCTION trg_rowcnt()
RETURNS TRIGGER LANGUAGE PLPGSQL AS
$func$
DECLARE
    v_cnt BIGINT;
BEGIN

    SELECT COUNT(*) INTO v_cnt FROM foo;

    RAISE NOTICE 'Table row count: %', v_cnt;

    IF (TG_OP = 'DELETE') THEN
        RETURN OLD;
    ELSE
        RETURN NEW;
    END IF;

END;
$func$;
```



# Triggers...

```
CREATE TRIGGER foo_trg_05
  BEFORE INSERT OR UPDATE OR DELETE
  ON foo
  FOR EACH ROW
  EXECUTE FUNCTION trg_rowcnt();

CREATE TRIGGER foo_trg_10
  BEFORE INSERT OR UPDATE OR DELETE
  ON foo
  FOR EACH ROW
  EXECUTE FUNCTION trg_slow(5);

CREATE TRIGGER foo_trg_15
  BEFORE INSERT OR UPDATE OR DELETE
  ON foo
  FOR EACH ROW
  EXECUTE FUNCTION trg_rowcnt();
```

```
-- Test script:
DELETE FROM foo;
SELECT now();
INSERT INTO foo VALUES (1, 1);
SELECT now();
DELETE FROM foo;
SELECT now();

-- Concurrency test statement:
INSERT INTO foo VALUES (1, 1);
```

# Check Triggers...

```
DELETE FROM foo;  
SELECT now();  
INSERT INTO foo VALUES (1, 1);  
SELECT now();  
DELETE FROM foo;  
SELECT now();
```

-----

```
DELETE 0
```

```
now
```

-----

```
2022-05-22 13:45:57.812879+02
```

```
psql:e.sql:76: NOTICE: Table row count: 0  
psql:e.sql:76: NOTICE: Table row count: 0  
INSERT 0 1
```

```
now
```

-----

```
2022-05-22 13:46:02.829487+02
```

```
psql:e.sql:78: NOTICE: Table row count: 1  
psql:e.sql:78: NOTICE: Table row count: 2  
DELETE 1
```

```
now
```

-----

```
2022-05-22 13:46:07.842845+02  
(1 row)
```

```
=# INSERT INTO foo VALUES (1, 1);  
NOTICE: Table row count: 1  
NOTICE: Table row count: 1  
INSERT 0 1
```

# Read Committed - projection example

```
CREATE OR REPLACE FUNCTION f_cnt() RETURNS BIGINT
LANGUAGE SQL AS
$func$
    SELECT count(*) FROM foo, pg_sleep(5);
$func$;

TRUNCATE TABLE foo; INSERT INTO foo VALUES (1, 1), (1, 1), (1, 1);
```

```
SELECT f_cnt() FROM foo;
```



```
INSERT INTO foo VALUES (1, 1);
```

```
\watch 5
```

# Read Committed - projection example

```
CREATE OR REPLACE FUNCTION f_cnt() RETURNS BIGINT
LANGUAGE SQL AS
$func$
    SELECT count(*) FROM foo, pg_sleep(5);
$func$;

TRUNCATE TABLE foo; INSERT INTO foo VALUES (1, 1), (1, 1), (1, 1);
```

```
SELECT f_cnt() FROM foo;
 f_cnt
-----
      3
      4
      5
(3 rows)
```

```
Time: 15014,130 ms (00:15,014)
```

```
INSERT INTO foo VALUES (1, 1);

\watch 5
```

# Read Committed - relations example

```
CREATE OR REPLACE FUNCTION f_cnt() RETURNS BIGINT
LANGUAGE SQL AS
$func$
    SELECT count(*) FROM foo, pg_sleep(5);
$func$;

TRUNCATE TABLE foo; INSERT INTO foo VALUES (1, 1), (1, 1), (1, 1);
```

```
SELECT * FROM foo, f_cnt();
```



```
INSERT INTO foo VALUES (1, 1);

\watch 5
```

# Read Committed - relations example

```
CREATE OR REPLACE FUNCTION f_cnt() RETURNS BIGINT
LANGUAGE SQL AS
$func$
    SELECT count(*) FROM foo, pg_sleep(5);
$func$;

TRUNCATE TABLE foo; INSERT INTO foo VALUES (1, 1), (1, 1), (1, 1);
```

```
SELECT * FROM foo, f_cnt();
 id | val | f_cnt
----+-----+-----
  1 |   1 |     3
  1 |   1 |     3
  1 |   1 |     3
(3 rows)
Time: 5006,045 ms (00:05,006)
```

```
INSERT INTO foo VALUES (1, 1);

\watch 5
```

# Read Committed - lateral example

```
CREATE OR REPLACE FUNCTION f_cnt() RETURNS BIGINT
LANGUAGE SQL AS
$func$
    SELECT count(*) FROM foo, pg_sleep(5);
$func$;

TRUNCATE TABLE foo; INSERT INTO foo VALUES (1, 1), (1, 1), (1, 1);
```

```
SELECT * FROM foo
  JOIN LATERAL f_cnt() ON TRUE;
 id | val | f_cnt
----+----+-----
  1 |  1 |     3
  1 |  1 |     3
  1 |  1 |     3
(3 rows)
Time: 5006,045 ms (00:05,006)
```

```
INSERT INTO foo VALUES (1, 1);

\watch 5
```

# Read Committed - lateral example 2

```
CREATE OR REPLACE FUNCTION f_cnt_param(int) RETURNS BIGINT
LANGUAGE SQL AS
$func$
SELECT count(*) FROM foo, pg_sleep(5);
$func$;

TRUNCATE TABLE foo; INSERT INTO foo VALUES (1, 1), (1, 1), (1, 1);
```

```
SELECT * FROM foo
  JOIN LATERAL f_cnt_param(foo.id) ON TRUE;
 id | val | f_cnt_param
----+----+-----
  1 |  1 |           3
  1 |  1 |           4
  1 |  1 |           5
(3 rows)
Time: 15011,754 ms (00:15,012)
```

```
INSERT INTO foo VALUES (1, 1);

\watch 5
```



# Repeatable read

```
SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

```
f_cnt
```

```
-----
```

```
3
```

```
3
```

```
3
```

```
(3 rows)
```

```
Time: 15011,280 ms (00:15,011)
```

```
id | val | f_cnt
```

```
---+-----+---
```

```
1 | 1 | 3
```

```
1 | 1 | 3
```

```
1 | 1 | 3
```

```
(3 rows)
```

```
Time: 5004,129 ms (00:05,004)
```

```
id | val | f_cnt
```

```
---+-----+---
```

```
1 | 1 | 3
```

```
1 | 1 | 3
```

```
1 | 1 | 3
```

```
(3 rows)
```

```
Time: 5001,980 ms (00:05,002)
```

```
id | val | f_cnt_param
```

```
---+-----+-----
```

```
1 | 1 | 3
```

```
1 | 1 | 3
```

```
1 | 1 | 3
```

```
(3 rows)
```

```
Time: 15012,739 ms (00:15,013)
```

# Read committed

- No dirty reads in PostgreSQL.
- Non-repeatable reads.
- Inconsistent reads (result of non-repeatable reads).
- Volatile functions (the default) are executed for each row and see changes committed by other tx.
- Triggers leverages on trigger functions and they are volatile (despite the syntax options available). For more details see [mailing list thread](#).

# Repeatable read as a rescue...

```
tril=# INSERT INTO FOO VALUES (1, 1);  
INSERT 0 1
```

```
tril=# DELETE FROM foo;  
DELETE 1
```

```
tril=# START TRANSACTION ISOLATION LEVEL  
REPEATABLE READ;
```

```
tril=# table foo;  
id | val  
----+-----  
 1 |   1
```

```
tril=# table foo;  
id | val  
----+-----  
 1 |   1
```

```
tril=# UPDATE foo SET val=val+1;  
ERROR:  could not serialize access due to  
concurrent delete
```

# Repeatable read coding impact

No lost changes as described in Read committed. [Nirvana...](#)

On the other hand [You can't have your cake and eat it](#) so it comes with some price:

```
ERROR: could not serialize access due to concurrent update
```

Your application have to be ready to handle these exceptions.

If you are using **postgres\_fdw**, your remote transactions are at [REPEATABLE READ](#) isolation level (unless local transaction is SERIALIZABLE), therefore don't be surprised by a serialization error even if local transaction is READ COMMITTED.

# Serializable

No anomalies are allowed.

Throughput is affected due to necessary locks used as an addition to snapshot isolation.

- serialization error might occur
- lock waits to prevent Read-only transaction anomaly
  - lock on transactionid

# Serializable

```
tril=# CREATE TABLE foo (cust_id int, val int);
tril=# CREATE TABLE bar (cust_id int, val int);
tril=# INSERT INTO foo values (1, 12);
tril=# INSERT INTO bar values (1, 12);
```

```
tril=# START TRANSACTION ISOLATION LEVEL
SERIALIZABLE;
```

```
tril=# TABLE foo;
  cust_id | val
-----+-----
         1 | 12
```

```
tril=# UPDATE foo SET val = 8 WHERE cust_id = 1;
tril=# SELECT foo.val + bar.val FROM foo JOIN bar
USING (cust_id);
?column?
-----
        20
```

```
tril=# COMMIT;
COMMIT
```

```
tril=# START TRANSACTION ISOLATION LEVEL
SERIALIZABLE;
```

```
tril=# table bar;
  cust_id | val
-----+-----
         1 | 12
```

```
tril=# UPDATE bar SET val = 8 WHERE cust_id = 1;
tril=# SELECT foo.val + bar.val FROM foo JOIN bar
USING (cust_id);
?column?
-----
        20
```

```
tril=# COMMIT;
```



# Serializable

```
tril=# CREATE TABLE foo (cust_id int, val int);
tril=# CREATE TABLE bar (cust_id int, val int);
tril=# INSERT INTO foo values (1, 12);
tril=# INSERT INTO bar values (1, 12);

tril=# START TRANSACTION ISOLATION LEVEL
SERIALIZABLE;

tril=*# TABLE foo;
  cust_id | val
-----+-----
         1 | 12

tril=*# UPDATE foo SET val = 8 WHERE cust_id = 1;
tril=*# SELECT foo.val + bar.val FROM foo JOIN bar
USING (cust_id);
?column?
-----
         20

tril=*# COMMIT;
COMMIT
```

```
tril=# START TRANSACTION ISOLATION LEVEL
SERIALIZABLE;

tril=*# table bar;
  cust_id | val
-----+-----
         1 | 12

tril=*# UPDATE bar SET val = 8 WHERE cust_id = 1;
tril=*# SELECT foo.val + bar.val FROM foo JOIN bar
USING (cust_id);
?column?
-----
         20

tril=*# COMMIT;
ERROR:  could not serialize access due to
read/write dependencies among transactions
DETAIL:  Reason code: Canceled on identification as
a pivot, during commit attempt.
HINT:  The transaction might succeed if retried.
```

Thank you for your attention







Photo by Annie Spratt on Unsplash

# MySQL

```
mysql> CREATE TABLE foo(id INT, val INT);
```

```
mysql> INSERT INTO foo VALUES (1, 1);
```

```
mysql> DROP TABLE FOO;
```

```
mysql> SELECT * FROM foo;
```

```
+-----+-----+  
| id   | val  |  
+-----+-----+  
|    1 |    1 |  
+-----+-----+
```

```
mysql> SELECT * FROM foo;  
ERROR 1146 (42S02): Table 'mysql.foo'  
doesn't exist
```

```
mysql> START TRANSACTION;
```

```
mysql> CREATE TABLE foo (id int, val int);
```

```
mysql> START TRANSACTION;
```

```
mysql> SELECT * FROM foo;
```



# MySQL

```
mysql> START TRANSACTION;
```

```
mysql> CREATE TABLE foo (id int, val int);
```

DDL commits

```
mysql> INSERT INTO foo VALUES (1, 1);
```

autocommit

```
mysql> START TRANSACTION;
```

```
mysql> SELECT * FROM foo;  
Empty set (0.00 sec)
```

```
mysql> SELECT * FROM foo;
```

???



```
mysql> START TRANSACTION;
```

```
mysql> CREATE TABLE foo (id int, val int);
```

DDL commits

```
mysql> INSERT INTO foo VALUES (1, 1);
```

```
mysql> START TRANSACTION;
```

```
mysql> SELECT * FROM foo;  
Empty set (0.00 sec)
```

```
mysql> SELECT * FROM foo;  
Empty set (0.00 sec)
```

MySQL implements Read Uncommitted isolation level (on InnoDB storage engine)

```
mysql> \h start transaction
Name: 'START TRANSACTION'
Description:
Syntax:
START TRANSACTION
    [transaction_characteristic [,
transaction_characteristic] ...]
```

```
transaction_characteristic: {
    WITH CONSISTENT SNAPSHOT
    | READ WRITE
    | READ ONLY
}
```

...  
The WITH CONSISTENT SNAPSHOT modifier does not change the current transaction isolation level

...  
The only isolation level that permits a consistent read is  
REPEATABLE READ.

```
mysql> SELECT @@transaction_ISOLATION;
```

```
+-----+
| @@transaction_ISOLATION |
+-----+
| REPEATABLE-READ        |
+-----+
```

```
mysql> commit;
```

```
mysql> SELECT * FROM foo;
```

```
+----+-----+
| id  | val  |
+----+-----+
|    1 |    1 |
+----+-----+
```

# MySQL repeatable read

```
mysql> CREATE TABLE foo (id int, val int);  
mysql> INSERT INTO foo VALUES (1, 1);
```



autocommit

```
mysql> DELETE FROM foo;  
Query OK, 1 row affected (0.02 sec)
```

```
mysql> START TRANSACTION;
```

```
mysql> SELECT * FROM foo;
```

id	val
1	1

```
mysql> SELECT * FROM foo;
```

id	val
1	1

continue from  
previous slide

# MySQL repeatable read

```
mysql> CREATE TABLE bar(id int, va int);  
Query OK, 0 rows affected (0.03 sec)
```

```
mysql> INSERT INTO bar VALUES (1, 1);  
Query OK, 1 row affected (0.01 sec)
```

```
mysql> SELECT * FROM bar;  
ERROR 1412 (HY000): Table definition has  
changed, please retry transaction
```

```
mysql> SELECT * FROM foo;
```

no rows

```
mysql> SELECT * FROM bar;
```

```
+-----+-----+  
| id   | va   |  
+-----+-----+  
|    1 |    1 |  
+-----+-----+
```



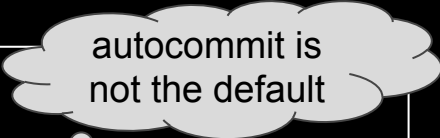
# Firebird ( Interbase )

```
CREATE TABLE foo(if INT, val INT);  
INSERT INTO foo VALUES (1, 1);
```

```
SQL> DROP TABLE FOO;  
Statement failed, SQLSTATE = 42000  
unsuccessful metadata update  
-object TABLE "FOO" is in use
```

```
SQL> DROP TABLE FOO;  
Statement failed, SQLSTATE = 42000  
unsuccessful metadata update  
-object TABLE "FOO" is in use
```

```
SQL> COMMIT;  
SQL> DROP TABLE FOO;
```



autocommit is  
not the default

```
SQL> SELECT * FROM foo;
```

```
SQL> COMMIT;
```

# Firebird ( Interbase )

```
SQL> SET TRANSACTION;  
SQL> CREATE TABLE foo (id int, val int);  
  
SQL> INSERT INTO foo VALUES (1, 1);
```

```
SQL> SET TRANSACTION;  
SQL> SELECT * FROM foo;  
  
SQL> SELECT * FROM foo;
```



# Firebird ( Interbase )

```
SQL> SET TRANSACTION;  
SQL> CREATE TABLE foo (id int, val int);  
  
SQL> INSERT INTO foo VALUES (1, 1);  
  
SQL> COMMIT;
```

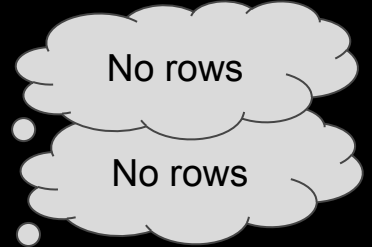
Defaults:

```
SET TRANSACTION  
  READ WRITE  
  WAIT  
  ISOLATION LEVEL SNAPSHOT;
```

The three isolation levels supported in Firebird are:

- SNAPSHOT
- SNAPSHOT TABLE STABILITY
- READ COMMITTED with two specifications (NO RECORD\_VERSION and RECORD\_VERSION)

```
SQL> SET TRANSACTION;  
  
SQL> SELECT * FROM foo;  
  
SQL> SELECT * FROM foo;  
  
SQL> SELECT * FROM foo;  
  
SQL> COMMIT;  
SQL> SELECT * FROM foo;
```



```
          ID          VAL  
=====  =====  
          1          1
```

[manual pages](#)

W E L C O M E

B A C K !



# Default transaction isolation levels

PostgreSQL:	<u>read committed</u>
MySQL:	<u>repeatable read</u>
Firebird:	<u>Snapshot</u>
Sybase:	Level 1 – <u>prevents dirty reads.</u>
MS SQL Server:	<u>read committed</u>
Oracle:	<u>read committed</u>
DB2:	<u>Cursor stability (CS)</u>
Informix:	<u>read committed</u>