Interesting Bugs Founded by DBStorm

We run *DBStorm* on several real-world classic open-source *DBMS*s, some of which have been deployed for commercial businesses for a long time. *DBStorm* detects 22 bugs. Here we present the details of five interesting bugs in pessimistic transaction mode exposed in *TiDB*.

```
CREATE TABLE t(a INT PRIMARY KEY, b INT);
INSERT INTO t(676, -5012153);
BEGIN TRANSACTION;--TID:739
UPDATE t SET b=-5012153 WHERE a=676;--TID:739
UPDATE t SET b=-852150 WHERE a=676;--TID:723✖
COMMIT;--TID:739
```

Listing 1. Dirty Write

Case Study 1: Dirty Write. In Listing. 1, transaction $TID = 739$ writes a record (i.e., a=676), and then another transaction $TID = 723$ also writes this record before 739 commits, which results in a dirty write. We find that the first update does not modify the record, leading to *TiDB* acquiring no lock, i.e., dirty write anomaly from the perspective of an application.

```
CREATE TABLE t(a INT PRIMARY KEY, b DOUBLE);
INSERT INTO t(3873, -1.123);
UPDATE t SET b=-0.386 WHERE a=3873;--TID:904
UPDATE t SET b=0.484 WHERE a=3873;--TID:907
SELECT b FROM t WHERE a=3873;
--TID:914, Result:{-0.386}✖
```

Listing 2. Inconsistent Read

Case Study 2: Inconsistent Read. In Listing. 2, transaction $TID = 914$ reads the record written by the first update $TID = 904$, but does not read the latest one written by the second update $TID = 907$, which violates the linearizability.

```
CREATE TABLE t(a INT PRIMARY KEY, b INT);
CREATE TABLE s(a INT PRIMARY KEY, b INT);
ALTER TABLE s ADD FOREIGN KEY(b) REFERENCES t(a));
INSERT INTO t(1, 2);
INSERT INTO s(2, 1);
BEGIN TRANSACTION;--TID:211
UPDATE t SET b=3 WHERE a=1;--TID:211
SELECT * FROM t, s WHERE t.a=s.b AND s.a>1
FOR UPDATE; --TID:324, Result:{2,1,2}✖
COMMIT;--TID:211
```

Listing 3. Incompatible Write Locks

Case Study 3: Incompatible Write Locks. In Listing. 3, transaction $TID = 211$ acquires a long write lock on record 1 in table t, and another concurrent transaction $TID = 324$ successfully reads record 1 in table t by *FOR UPDATE* statement, which violates the mutual exclusion between write locks. It is worth noting that 324 accesses record 1 of table t through $join$ operator. Before accessing the record 1 of table t, TiDB forgets the lock acquisition, leading to this bug.

```
CREATE TABLE t(a INT PRIMARY KEY, b INT);
BEGIN TRANSACTION;--TID:242
UPDATE t SET b=3 WHERE a=1;--TID:242
INSERT INTO t VALUES(1,5);--TID:432, Status:
  blocking✖
COMMIT;--TID:242
```

Listing 4. Over Locking

Case Study 4: Over Locking. Most production-level *DBMS*s take range locks or *MVCC* to avoid phantom. In Listing. 4, transaction TID=242 updates a non-exist record and locks it. However, *TiDB* provides only *MVCC* to avoid phantom, and does not provide a range lock mechanism. This error lowers the insertion performance of concurrent transactions due to blocking.

```
CREATE TABLE t(a INT PRIMARY KEY, b INT);
CREATE TABLE s(a INT PRIMARY KEY, b INT);
ALTER TABLE s ADD FOREIGN KEY(b) REFERENCES t(a));
INSERT INTO t(1, 2);
INSERT INTO s(2, 1);
DELETE FROM s WHERE a=2;--TID:213
BEGIN TRANSACTION;--TID:412
INSERT INTO s VALUES(2,3);--TID:412
SELECT * FROM t WHERE a=2;
--TID:412, Result:{2,1},{2,3}✖
```

Listing 5. A Query that Returns two versions

Case Study 5: A Query that Returns two versions. According to linearizability, a query should fetch the version of a record that creates just before the query, i.e., the run-time latest version. In Listing. , transaction $TID = 412$ returns two versions for a record. One is the version written by 412 itself, and the other is the deleted version, which should not be available. We report this problem to *TiDB* and confirmed that it was a known bug. The reason for this bug is that the scan operator in *TiDB* handles integer and non-integer types incorrectly in the unique index.

```
CREATE TABLE t(a FLOAT PRIMARY KEY AUTO_INCREMENT,b
    INT);
INSERT INTO t(b) VALUES(8784);
INSERT INTO t(b) VALUES(23371);
INSERT INTO t(b) VALUES(37958);
SELECT SUM(pk) FROM t WHERE b < 61883;--Result:{9}✖
```

From the above five cases, we have learned the following lessons:

- Stochastic testing techniques have great advantages for finding bugs. It can trigger more boundary conditions, especially with no prior knowledge of the source code of the software.
- Handling contention is the main task for transaction processing. Both α and β validity decide the contention in a workload. Thus valid workload is critical for testing transaction processing.