Interesting Bugs Found by Leopard

Keqiang Li
East China Normal University, Shanghai, China
kqli@stu.ecnu.edu.cn

Leopard has successfully discovered 20 transactional bugs from real-world production-level $DBMS$s, including 1 bugs in MySQL, 2 bugs in PostgreSQL, 11 bugs in TiDB, 2 bugs in OpenGauss, and 2 bugs in a series of commercial $DBMS$s. In [**?**], we list all bugs having received positive feedbacks. Here we demonstrate the details of five interesting bugs in pessimistic transaction mode in *TiDB*.

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

Listing 1. Dirty Write

Case Study 1: Dirty Write. In Listing. 1, transaction $t = 739$ writes a record (i.e., a=676), and then another transaction $t = 723$ also writes this record before 739 commits, which results in a dirty write [**?**]. We find that the first update does not really modify the record, leading to *TiDB* acquiring no lock, i.e., dirty write anomaly from perspective of application. This bug will not be found by the traditional verification method, e.g., the *Consistency Tests* in *TPC-C*.

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

Listing 2. Inconsistent Read

Case Study 2: Inconsistent Read. The verification based on traces can provide the timestamps of invocation and completion of each statement. Therefore, *Leopard* can order the statements executed in the database. In Listing. 2, transaction $t = 914$ reads the record written by the first update $t = 904$, but does not read the latest one written by the second update $t = 907$, which violates the linearizability.

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

Listing 3. Incompatible Write Locks

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

```
CREATE TABLE r(a INT PRIMARY KEY, b INT);
BEGIN TRANSACTION;—t:242
UPDATE r SET b=3 WHERE a=1;—t:242
INSERT INTO r VALUES(1,5);—t:432, Status:blocking
✖
COMMIT;—t: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 t=242 updates a non-exist record and locks it. However, *TiDB* provides only *MVCC* to avoid phantom, and does not provide range lock mechanism. This error behavior lowers down the insertion performance of concurrent transactions due to blocking.

```
CREATE TABLE r(a INT PRIMARY KEY, b INT);
CREATE TABLE s(a INT PRIMARY KEY, b INT);
ALTER TABLE s ADD FOREIGN KEY(b) REFERENCES r(a));
INSERT INTO r(1, 2);
INSERT INTO s(2, 1);
DELETE FROM s WHERE a=2;—t:213
BEGIN TRANSACTION;—t:412
INSERT INTO s VALUES(2,3);—t:412
SELECT * FROM r WHERE a=2;
—t: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. 5, transaction $t = 412$ returns two versions for a record. One is the version written by 412 itself, and the other is the deleted version, which should be fetched. 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.

From the above five cases, we have following lessons learned: *bug-exposed trace* is a useful approach that provides a general way to verify isolation levels for various workloads under black-box.