Practice: Using Lock-Free Reservations

Practice Target

In this practice, you will enable and observe **Lock-Free Reservations**. You will create a simple concert ticket booking scenario with two tables, demonstrate traditional blocking behaviour.

Practice Overview

In high level, you will perform the following tasks:

1. **Create and populate the CONCERTS and TICKET_BOOKINGS tables** Build a scenario where multiple sessions can update the same concert row.
2. **Demonstrate blocking sessions**
   Show how concurrency leads to blocking when two sessions update the same row at the same time.
3. **Enable and test Lock-Free Reservations**
   Convert a column to RESERVABLE, preventing blocking.
4. **Explore Lock-Free Reservation restrictions**
   Demonstrate important usage constraints for reservable columns.

Creating the Demonstration Environment

In this first step, you will create two tables in the **HR** schema to simulate a concert ticket system. The *CONCERTS* table tracks how many tickets remain (*tickets_left*), while the *TICKET_BOOKINGS* table logs each booking attempt.

1. Start **SQL Developer** and connect to the database as **HR**.

2. Create the **CONCERTS** and **TICKET_BOOKINGS** tables follows:

```
CREATE TABLE concerts (
   concert_id   NUMBER PRIMARY KEY,
concert_name VARCHAR2(100),    tickets_left
NUMBER
);

INSERT INTO concerts (concert_id, concert_name, tickets_left)
VALUES (1, 'Rock Night', 50);

INSERT INTO concerts (concert_id, concert_name, tickets_left)
VALUES (2, 'Jazz Evening', 100);

INSERT INTO concerts (concert_id, concert_name, tickets_left)
VALUES (3, 'Pop Festival', 200);

COMMIT;

CREATE TABLE ticket_bookings (
   booking_id     NUMBER GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
   concert_id     NUMBER,
tickets_booked NUMBER,
CONSTRAINT concerts_fk
      FOREIGN KEY (concert_id) REFERENCES concerts (concert_id)
);
```

3. Implement a trigger to reduce *tickets_left* in **CONCERTS** whenever a new booking is inserted:

```
CREATE OR REPLACE TRIGGER ticket_bookings_trg
AFTER INSERT ON ticket_bookings
FOR EACH ROW
BEGIN
   UPDATE concerts c
      SET c.tickets_left = c.tickets_left - :NEW.tickets_booked
    WHERE c.concert_id = :NEW.concert_id;
END;
/
```

Demonstrating Blocking Sessions

By default, concurrent updates to the same row can block each other. In this section, you will implement the scenario of booking tickets for the same concert from two sessions simultaneously.

4. Open two different sessions in Putty and connect to the vm as `oracle`. Each session makes a reservation as follows:

5. In **Session 1**:

```
sqlplus hr/oracle@freepdb1

INSERT INTO ticket_bookings (concert_id, tickets_booked) VALUES
(1, 10);
```

6. In **Session 2**:

```
sqlplus hr/oracle@freepdb1

INSERT INTO ticket_bookings (concert_id, tickets_booked) VALUES
(1, 6);
```

Notice that **Session 2** will block until **Session 1** issues a commit or rollback because both are updating the same row in *concerts*.

7. Commit **Session 1**, then observe that **Session 2** completes immediately. Commit **Session 2** as well. Finally, check *tickets_left*:

```
-- Session 1
COMMIT;

-- Session 2
COMMIT;
col CONCERT_NAME for a20
SELECT * FROM concerts WHERE concert_id = 1;
```

Enabling and Testing Lock-Free Reservations

Next, you will configure the *tickets_left* column as **RESERVABLE**, ensuring concurrent updates do not block each other. Instead, they reserve updates that only apply on commit.

8. In SQL Developer, reset your tables for a fresh test:

```
TRUNCATE TABLE ticket_bookings;

UPDATE concerts
   SET tickets_left = CASE concert_id
                          WHEN 1 THEN 50
                          WHEN 2 THEN 100
                          WHEN 3 THEN 200
                       END;

COMMIT;
```

9. Alter *tickets_left* to become **RESERVABLE**.

   Technically, creating a CHECK constraint is not needed for a RESERVABLE column. However, practically we should create a CHECK constraint on it according to the business **requirements** to avoid inserting invalid values into the column.

```
ALTER TABLE IF EXISTS concerts
   MODIFY (tickets_left RESERVABLE);
ALTER TABLE IF EXISTS concerts ADD CONSTRAINT tickets_left_chk
CHECK (tickets_left>=0);
```

10. Identify the reservation journal table and take a copy of its name. The journal typically has a name like *SYS_RESERVJRNL_####*.

    Oracle Database stores “reserved” updates in an internal journal table and applies them only on commit. Each session can query this table to see pending reservations made by the session.

```
SELECT object_name
FROM   user_objects
WHERE  object_type = 'TABLE'
AND    object_name LIKE 'SYS_RESERVJRNL_%';
```

11. **Describe** the reservation journal table.

    The table includes the primary key column (CONCERT_ID), the operation on the TICKETS_LEFT column (TICKETS_LEFT_OP) and the reserved value for the operation on the TICKETS_LEFT column (TICKETS_LEFT_RESERVED).

```
DESC SYS_RESERVJRNL_####
```

12. In the two sessions, insert bookings for the same concert without committing.

```
-- Session 1
INSERT INTO ticket_bookings (concert_id, tickets_booked)
VALUES (1, 10);

-- Session 2
INSERT INTO ticket_bookings (concert_id, tickets_booked)
VALUES (1, 5);
```

As a result of having a RESERVABLE column in the table, no blocking occurs; the second session is not forced **to** wait.

13. Before committing, in each session, query the **CONCERTS** table and note that *tickets_left* remains 50, which means it was not changed.

```
column CONCERT_NAME for a20
SELECT * FROM concerts WHERE concert_id = 1;
```

14. In each session, check the reservation journal table.

    Each session sees the number of tickets reserved by itself. It does not see the number of tickets reserved by the other sessions.

```
select ora_stmt_type$,
CONCERT_ID,
       TICKETS_LEFT_OP,
TICKETS_LEFT_RESERVED from
SYS_RESERVJRNL_###;
```

15. Open Putty and login to the vm as `oracle`. Create a third SQL*Plus session as HR.

```
sqlplus hr/oracle@freepdb1
```

16. Try reserving 40 tickets.

```
INSERT INTO ticket_bookings (concert_id, tickets_booked) VALUES
(1, 40);
```

    You will receive an error like the following:

```
ERROR at line 1:
ORA-02290: check constraint (HR.TICKETS_LEFT_CHK) violated ORA-06512:
at "HR.TICKET_BOOKINGS_TRG"
```

    This is expected because session 1 and session 2 reserved 15 tickets out of total 50 tickets. This means maximum 35 tickets can be reserved. This session tried to reserve 40 tickets. Oracle database needs to refer to the Journal table to maintain the constraint.

17. Commit both sessions (in any order), then check *tickets_left*.

```
-- Session 1
COMMIT;

-- Session 2
COMMIT;

SELECT * FROM concerts WHERE concert_id = 1;
```

You will see *tickets_left* reduced by the combined total of both booking statements. 35 tickets remained.

Checking the Lock-Free Reservation Restrictions

In the following steps, you will demonstrate the constraints enforced on tables containing RESERVABLE columns. These constraints are essential for maintaining data integrity during operations on lock-free tables.

18. Attempt a direct assignment:

    Only increment or decrement (+ or -) is allowed on the reservable columns.

```
UPDATE concerts
   SET tickets_left = 999
 WHERE concert_id = 2;
```

You **will** get an error (*ORA-55746*) indicating only **+** or **-** operations are supported.

19. Execute the following UPDATE statement.

```
UPDATE concerts
   SET tickets_left = tickets_left - 1;
```

The statement will fail with an error (*ORA-55732*) because you must include all primary key columns in **the** WHERE clause.

The primary key must be referenced. An update without specifying CONCERT_ID in the WHERE clause is not allowed.

20. Execute the following statements in session1 and session2.

```
-- Session 1: update but do not commit
UPDATE concerts
   SET tickets_left = tickets_left - 5
 WHERE concert_id = 3;

-- Session 2: try to delete same row
DELETE FROM concerts WHERE concert_id = 3;
```

Only after Session 1 commits or rolls back can you delete that row.

No delete if there is an outstanding reservation. If a transaction has not yet committed a reservation, a delete attempt on that row fails (*ORA-55754*) until the transaction ends.

21. Try dropping the table.

```
DROP TABLE concerts PURGE;
```

You **will get *ORA-55764*. You c**annot drop a table with a reservable column without reverting.

To drop the table, you must perform the following:

```
 ALTER TABLE IF EXISTS concerts MODIFY (tickets_left NOT RESERVABLE);
 DROP TABLE IF EXISTS concerts PURGE;
```

22. Clean up existing objects if they exist

```
DROP TRIGGER IF EXISTS ticket_bookings_trg;
DROP TABLE   IF EXISTS ticket_bookings PURGE;
ALTER TABLE  IF EXISTS concerts MODIFY (tickets_left NOT
RESERVABLE); DROP TABLE   IF EXISTS concerts PURGE;
```

Summary

In this practice, you built a simple concert ticket booking scenario and showed how multiple users could block each other when updating the same row. After enabling **Lock-Free Reservations** on the *tickets_left* column, you observed non-blocking inserts in separate sessions—pending reservations were only applied upon commit. You also learned to check the reservation journal table to see those reservations before commit, and discovered important usage constraints for reservable columns: **only + or -** operations, **must** specify the primary key, **cannot** delete or drop while reservations exist, and more.

This feature ensures concurrency on frequently updated numeric columns can be handled efficiently without blocking in Oracle Database 23ai.