12. Transaction Processing

Most of content here is taken from PostgreSQL docuentation

Concurrency Control Techniques in PostgreSQL

- PostgreSQL maintains data consistency using a multi-version model (Multi-version Concurrency Control, MVCC)
- MVCC minimizes lock contention
- MVCC allows high concurrency even during significant database read/write activity. MVCC specifically offers behavior where "reads never block writes, and writes never block reads"**.**
- PostgreSQL also supports table and row-level locking for situations where MVCC does not adapt well.

Isolation levels in PostgreSQL

- In PostgreSQL, you can request any of the four standard transaction isolation levels.
- But internally, there are only two distinct isolation levels, which correspond to the levels Read Committed and Serializable.
- When you select the level Read Uncommitted you really get Read Committed, and when you select Repeatable Read you really get Serializable
- Read Committed is the default isolation level in PostgreSQL
- PostgreSQL uses “snapshot isolation”, and therefore its serializable isolation level is not true serializable.

SQL commands in PostgreSQL

- Start Transaction: begin transaction or start transaction
- Commit transaction: commit, commit transaction, end transaction.
- Abort transaction: rollback, or rollback transaction.
- No support for AUTO COMMIT – sensed by context of run

Read Committed Isolation level

- A “select query” takes snapshot of data item at the beginning of statement; any update during the execution of statement is not seen by the query
- However if data is queried again even in same transaction, it sees updates from committed concurrent transactions, and the query result can be different (problem of non-repeatable read)

- UPDATE commands behave the same as SELECT in terms of searching for target rows; they will only find target rows that were committed as of the start time of the command; if such a target row have already been updated (or deleted or "locked") by another concurrent transaction by the time the transaction finds them.
- If the first updater rolls back, then second updater can proceed with updating the originally found row.
- If the first updater commits, the second updater will ignore the row if the first updater deleted it, otherwise it will attempt to apply its operation to the updated version of the row.
- The search condition of the command (the WHERE clause) is re-evaluated to see if the updated version of the row still matches the search condition. If so, the second updater proceeds with its operation, starting from the updated version of the row.
- Phantom rows are also seen by second updater.
- Consider example below; following concurrent execution of two transactions-

```
T1: begin;
T1: update account set balance = balance+5000 where acno='101';
T2: begin;
T2: update account set balance = balance-3000 where acno='101';
        //T2 should be waiting
T1: commit;
T2: commit;

select * from account where acno='101';
```

should show balance 22000 (if initial balance is 20000)

Note: If T2 runs in serializable isolation level then T2 will have to be rolled back as in read repeatable T2 does not see consistent view of data items.
Below is differently written version of above transactions (it basically simulates read of T2 before write of T1) and demonstrates problem due to non-repeatable read! The problem simply because there is a cycle in conflicting operations; setting isolation level of T2 to serializable will abort the T2

```
T1: update account set balance = balance+5000 where acno='101';
T2: update account set balance =
    (select balance from account where acno= '101' ) -3000 where acno='101';
T1: commit;
T2: commit;
```

If you execute T2 in above example in SERIALIZABLE isolation, you get following message for T2

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

==> What is benefit and cost of running a transaction at lower isolation level?

Benefit: increased concurrency (otherwise T2 would have terminated)
Cost: non-serializable execution (non-repeatable read), however this is not a problem if query is written in more natural way as the case above)

Serializable Isolation level

- When a transaction is on the serializable level, a SELECT query sees only data committed before the transaction began; it never sees either uncommitted data or changes committed during transaction execution by concurrent transactions.
- This is different from Read Committed in that the SELECT sees a snapshot as of the start of the transaction, not as of the start of the current query within the transaction. Thus, successive SELECT commands within a single transaction always see the same data, i.e. Has Repeatable reads.
- UPDATE and DELETE commands behave the same as SELECT in terms of searching for target rows: they will only find target rows that were committed as of the transaction start time.
- However, such a target row might have already been updated (or deleted or locked) by another concurrent transaction by the time it is found.
- In this case, the serializable transaction will wait for the first updating transaction to commit or roll back (if it is still in progress).
- If the first updater rolls back, then its effects are negated and the serializable transaction can proceed with updating the originally found row.
- But if the first updater commits (and actually updated or deleted the row) then the serializable transaction will be rolled back with the message because a serializable transaction cannot modify changed by other transactions after the serializable transaction began (first updater wins and other gets rolled back)

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

Serializable Isolation in PostgreSQL versus True Serializability

- PostgreSQL SERIALIZABLE isolation level is not truly *serializable*
- Consider example below, where two transactions T1 and T2 are executing at serializable isolation level; this will not give you the expected behavior

```
T1: update employee set salary =
    (select salary from employee where ssn = '101') where ssn = '105';
T2: update employee set salary =
    (select salary from employee where ssn = '105') where ssn = '101';
```

- This is basically the problem of "snapshot" isolation itself; you can however solve this problem by using explicit locks, and in this case first commit T1 wins and later committer rolls back.

```
T1: update employee set salary =
      (select salary from employee where ssn = '101' FOR UPDATE) where ssn = '105';
T2: update employee set salary =
     (select salary from employee where ssn = '105' ) where ssn = '101';
T1: commit;
T2: commit;
```

Another example

class	value
1	10
1	20
2	100
2	200

- Suppose you have a table **mytable** with snapshot as shown here. Concurrent execution of following transactions T1 and T2 (at serializable isolation level) will not be equivalent to any serial schedule in postgreSQL

```
T1: insert into mytable values(1, (SELECT SUM(value) FROM mytable WHERE class = 2));
T2: insert into mytable values(2, (SELECT SUM(value) FROM mytable WHERE class = 1));
```

Since version 9.1, postgresql implements Serializable Snapshot Isolation (SSI) and you may not experience "skew write"!

Request for locks

- Request for exclusive locks. Its general syntax is-
  SELECT … FOR UPDATE [OF *table_name*] [NOWAIT]
- This prevents them from being modified or deleted by other transactions until the current transaction ends. That is, other transactions that attempt UPDATE, DELETE, or SELECT FOR UPDATE of these rows will be blocked until the current transaction ends.
- Also, if an UPDATE, DELETE, or SELECT FOR UPDATE from another transaction has already locked a selected row or rows, SELECT FOR UPDATE will wait for the other transaction to complete, and will then lock and return the updated row (or no row, if the row was deleted)
- Request for share locks, its general syntax is-
  SELECT … FOR SHARE [OF *table_name*] [NOWAIT]
- Other transactions can perform SELECT … FOR SHARE, but not SELECT … FOR UPDATE

- To prevent the operation from waiting for other transactions to commit, use the **NOWAIT** option. **SELECT FOR UPDATE NOWAIT** reports an error, rather than waiting, if a selected row cannot be locked immediately.
- Note that **NOWAIT** applies only to the row-level lock(s)