Transaction Processing
and
Concurrency Control
Tutorial

SWEN 304

Trimester 2, 2017

Engineering and Computer Science

TE WHARE WĀNANGA O TE ŪPOKO O TE IKA A MĀUI
VICTORIA
UNIVERSITY OF WELLINGTON

Plan for Concurrency Control Tut

- Transaction support in SQL
  - BEGIN;
  - COMMIT;
  - SET TRANSACTION...;
- Support of isolation levels in PostgreSQL
  - Read committed
  - Serializable
- Explicit locking in PosgreSQL
  - LOCK TABLE command
  - Lock modes
- Avoiding unrepeatable read and lost update
  - SELECT... FOR UPDATE;
  - A disciplined approach
- PostgreSQL and dead-locks

Transaction Support in SQL

- Until now, we considered interactive SQL run from UNIX shell prompt
- SQL processor treats each SQL statement as a separate transaction with all or nothing (atomic) property, although there were no explicit Begin_Transaction and Commit (or Roll-back) statements
- Transaction initiation and Commit are done implicitly when particular SQL statements are encountered

Two Equivalent SQL Statements

```
INSERT INTO Student VALUES (7007,
'James', 'Bond', 'Comp');
```

```
BEGIN;
INSERT INTO Student VALUES (7007,
'James', 'Bond', 'Comp');
COMMIT;
```

- The only difference between these two INSERT statements is that the first one is implicitly and the other is explicitly wrapped into BEGIN…COMMIT statements

SQL Support of Transactions

- Using `BEGIN…COMMIT` to wrap a single SQL statement does not make much sense
- All SQL statements between a `BEGIN…  COMMIT` represent components of a single transaction with an (atomic) all or noting property (read committed)
- All locks obtained inside a `BEGIN…COMMIT` wrap are retained till the `COMMIT` point, when they are released
- Using `BEGIN…COMMIT` to wrap a single MULTISTATEMENT transaction does make a lot of sense

First Multi Statement Transaction

- No `BEGIN`…`COMMIT` wrap:

```
UPDATE Account
SET Balance = Balance – 100
WHERE AccountNo =1111;
// At that point DBMS stopped working for some
reason and the second update is not executed
UPDATE Account
SET Balance = Balance + 100
WHERE AccountNo = 2222;
```

- The net result is that $100 were lost

Second Multi Statement Transaction

- The same transaction as before, but with a `BEGIN…COMMIT` wrap:

```
BEGIN;
UPDATE Account
SET Balance = Balance – 100
WHERE AccountNo =1111;
// At that point DBMS stopped working for some
//reason and the second update and commit are not
//executed
UPDATE Account
SET Balance = Balance + 100
WHERE AccountNo = 2222;
COMMIT;
```

- The DBMS will roll-back the transaction since it did not reach the COMMIT point
- So, $100 were NOT lost

Transaction Oriented SQL Statements

- `BEGIN` (PostgreSQL specific)
- `SET TRANSACTION {READ ONLY | READ WRITE} DIAGNOSTIC SIZE n ISOLATION LEVEL` <*isolation*>
- `LOCK [TABLE]` <*name*> `IN` <*lock_mode*> (Postgre SQL specific)
- `CHECKPOINT` (PostgreSQL specific)
- `COMMIT`
- `ROLLBACK`

Support of Isolation Levels in PostgreSQL

- PostgreSQL really supports:
  - READ COMMITED (default) and
  - SERIALIZABLE

  isolation levels
- Unlike SQL/92, some PostgreSQL statements acquire locks automatically regardless of an isolation level:
  - The conventional SELECT statement does not acquire any lock on data but on schema constructs,
  - INSERT, DELETE, UPDATE statements acquire an exclusive lock on the rows selected,
  - and there are also some other statements that acquire particular locks

Read Committed in PostgreSQL

- A SELECT statement sees only data committed before the query is issued, so different queries inside the same transaction can see different database states
- If a row is locked exclusively by T_1, and a statement in T_2 wants to select this row for update or delete, T_2 will wait until T_1 ends
- T_2 will be applied on the database state defined by the outcome of T_1
- The isolation level READ COMMITTED prevents Dirty Read Transaction Anomaly to occur

Read Committed and SELECT

T_1	T_2
BEGIN; SELECT COUNT(*) FROM *Grades* WHERE *StudId=7007*; <u>count</u> 3 (1 row)	

	BEGIN;
	INSERT INTO *Grades* VALUES (*7007*, 'C305', 'A+');
	`INSERT 1`

SELECT COUNT(*) FROM *Grades* WHERE *StudId=7007*; <u>count</u> 3 (1 row)	
	COMMIT;
SELECT COUNT(*) FROM *Grades* WHERE *Stuld=7007*; <u>count</u> 4 (1 row) COMMIT;	

Read Committed and UPDATE

T_1	T_2
BEGIN;	
DELETE FROM *Grades* WHERE *StudId=7007*;	
`DELETE 3`	
	BEGIN;
	UPDATE *Grades* SET *Grade*='A+' WHERE StudId=7007;
COMMIT;	
	`UPDATE 0`
	COMMIT;

Read Committed and UPDATE

T$_1$	***T***$_2$
BEGIN;	
SELECT NoOfPts FROM *Student* WHERE *StudId=7007*;	
`NoOfPts` ------------- 165	BEGIN;
	UPDATE Student SET NoOfPts = NoOfPts + 15 WHERE *StudId=7007*;
UPDATE Student SET NoOfPts = NoOfPts – 15 WHERE *StudId=7007*; `UPDATE 1` COMMIT;	
	`UPDATE 1`
	SELECT NoOfPts FROM *Student* WHERE *StudId=7007*; `NoOfPts` ------------- 165 COMMIT;

Serializable in PostgreSQL

- Isolation level *serializable* emulates serial transaction execution
  - So, no explicit locks needed
- Within a serializable transaction a SELECT statement **does not** see any changes made by other committed transactions
- If a row is ***locked exclusively*** by T_1, and a statement in T_2 wants to select this row for update or delete, T_2 will wait until T_1 ends
- If T_1 commits T_2 will fail and will have to be retried
- If T_1 rollbacks T_2 will proceed

Serializable and SELECT

T_1	T_2
BEGIN;	
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;	BEGIN;
SELECT COUNT(*) FROM *Grades* WHERE *StudId=7007*; count 3 (1 row)	
	INSERT INTO *Grades* VALUES (*7007*, ‘C305’, ‘A+’); `INSERT 17003 1`
	COMMIT;
SELECT COUNT(*) FROM *Grades* WHERE *StudId=7007*; count 3 (1 row)	
COMMIT;	

Serializable and UPDATE

T_1	T_2
BEGIN;	
DELETE FROM *Grades* WHERE *StudId=7007*;	
`DELETE 3`	
	BEGIN; SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
	UPDATE *Grades* SET *Grade*='A+' WHERE *StudId*=7007;
COMMIT;	
	`ERROR: Can't serialize access due to concurrent update`
	COMMIT;

Explaining Behavior of Serializable

- The other update transaction fails if the first succeeds, because the other will read the stale value of a database item
- Contrary to isolation level Read Committed, after the first transaction finishes successfully and releases the locks, the second transaction reads the database state as it was when it started to execute
- If the first transaction fails, DBMS will rollback its changes against the database and the second transaction will read the correct database state (since it will be the same as it was when it started)

Explicit Locking in PostgreSQL

- PostgreSQL offers a LOCK TABLE command to fine tune the performance of a transaction program

- LOCK TABLE statement offers much more lock modes than we discussed in lectures

PostgreSQL LOCK Statement

```
LOCK [TABLE] <name> IN <lockmode> MODE;
```

```
<lockmode> :

  ACCESS SHARE | ROW SHARE | ROW
  EXCLUSIVE | SHARE UPDATE EXCLUSIVE |
  SHARE | SHARE ROW EXCLUSIVE |
  EXCLUSIVE | ACESS EXCLUSIVE
```

- It is not recommended to use it in Project 2

Semantics of the LOCK Statement

- All lock modes are **table-level** locks – the names are historical
- The only real difference between one lock mode and another is the set of lock modes with which each conflicts
  - If two lock modes do not conflict, both can be simultaneously acquired by different transactions on the same table
  - If two lock modes conflict, they can't be simultaneously acquired by two different transactions on the same table
- All lock modes, except `ACCESS EXCLUSIVE` allow concurrent reads
- Only particular lock modes prevent concurrent writes

Automatic Locks

- Some locks are acquired automatically by certain SQL statements
  - `SELECT and ANALYZE acquire ACCESS SHARE`
  - `SELECT…FOR UPDATE; acquires ROW SHARE on rows selected and ACCESS SHARE locks on rows referenced by rows selected in addition`
  - `UPDATE, DELETE, and INSERT acquire ROW EXCLUSIVE on rows targeted and ACCESS SHARE lock on rows referenced by rows targeted`
  - `CREATE INDEX acquires SHARE lock on the whole table`
  - `ALTER TABLE, DROP TABLE, REINDEX, CLUSTER, and VACUUM FULL acquire ACCESS EXCLUSIVE lock`

Effects of Lock Modes

Lock Mode	Effects	Conflicts	Acquired
ACCESS SHARE	Allows reads Allows writes	AE	SELECT, ANALYZE
ROW SHARE (RS)	Allows reads Allows writes, except to rows selected	AE	SELECT…FOR UPDATE;
ROW ECLUSIVE (RE)	Allows reads Allows writes, except to rows targeted	S, SRE, AE	UPDATE, DELETE, INSERT
SHARE (S)	Protects table from data changes	RE, SUE, SRE, AE	CREATE INDEX
SHARE ROW EXCLUSIVE (SRE)	Allows reads, prevents writes	RE, SUE, S, SRE, AE	
ACCESS EXCLUSIVE (AE)	Exclusive lock Prevents SELECT	ALL MODES	ALTER TABLE DROP TABLE

READ COMMITTED and Anomalies

- PostgreSQL default locking in the isolation level READ COMMITTED does not provide protection from:
  - Unrepeatable Read, and
  - Phantom record

  transaction anomalies, since
  - SELECT statement does not issue (practically) any lock request,
    - Hence unrepeatable, and
  - UPDATE, DELETE, and INSERT lock rows targeted, and rows referenced by rows targeted, only,
    - Hence phantom record

Avoiding Unrepeatable Read and Lost Update

- A good way to achieve a transaction program that is safe against dirty read and update anomalies with a good performance is to use
  - **`SELECT … FOR UPDATE` SQL command, and**
  - **A disciplined approach to the design of a transaction program**

- Use of **`SELECT … FOR UPDATE;`** is recommended for Project 2
- To avoid all anomalies, one may use (not recommended for Project 2)
  - `LOCK [TABLE] <name> IN ACCESS EXCLUSIVE MODE`

Row Level Locks

- SQL update commands acquire locks on rows targeted
  - The targeted rows are altered,
  - Reads of targeted rows are permitted,
  - Writes of targeted rows are forbidden

```
SELECT… FROM <table> WHERE <condition> FOR
UPDATE;
```

  - Doesn't alter the rows selected,
  - Acquires an exclusive lock on rows selected, and thus protects them from being written till the end of the transaction,
  - Allows "plain" `SELECT` on the tuples locked
- If a concurrent transaction issues a `SELECT. . .FOR UPDATE;` statement that selects any of already locked tuples, it will have to wait until the first transaction ends

A Disciplined Approach

- If a transaction is “read-only” use “plain” `SELECT…` to read data
- If a transaction is “read-write” issue `SELECT. . . FOR UPDATE;` commands for all tuples the transaction should either update or delete before any read, update or delete of these tuples
- This way, if a transaction T_1 locks a tuple t for update no other transaction will be able to acquire an exclusive lock on t until T_1 terminates

PostgreSQL and Dead-Locks

- PostgreSQL detects dead lock situations and roll-backs at least one of the transactions involved automatically
- To avoid dead-locks one:
  - May use `LOCK TABLE` commands to obey to *conservative two phase locking* protocol rule, or
  - Should impose an ordering on database items and always issue statements that require exclusive locks according to that ordering

Phantom Record

- A transaction T_1 locks database items that satisfy certain selection condition and updates them
- During that update, another transaction T_2 inserts a new item that satisfies the same selection condition
- After the update, we suddenly discover the existence of a database item that has not been updated although it should have been (since it satisfies the selection condition)
- The cause: the transaction T_1 locked the selected items only

Phantom Record Example

- Consider

  Stud_Pap_Assig(<u>StudentID, PapID, AssignmentNo,</u> NoOfMarks)

- Transaction T_1:
  - Makes an exclusive lock on tuples where *StudentID* = *7007*,
  - Reads corresponding *Stud_Pap_Assig* tuples,
  - Sums assignment marks, and
  - Displays the result
- In the meantime transaction T_2 inserts a new record (*7007*, *COMP302*, *Assig4*, *100*) into the database and the result computed is wrong

A remark

- PostgraSQL introduced Repeatable Read in Version 9.1

Isolation Level	Dirty Read	Non-repeatable Read	Phantoms
READ UNCOMMITTED	Possible	Possible	Possible
READ COMMITTED	Never occurs	Possible	Possible
REPEATABLE READ	Never occurs	Never occurs	Possible
SERIALIZABLE	Never occurs	Never occurs	Never occurs