Midterm Grading

Name:

**Task 1**: Autocommit

|  |  |  |  |
| --- | --- | --- | --- |
| Expected Result | Possible Points | Your  Points | Remarks |
| 1. Rows 1 and 2 are inserted because of autocommit: each statement is treated as transaction. rows 3-5 are not inserted because transaction is rolled back because of duplicate key error. 1p – correct answer 2p – correct explanation | 3 |  |  |
| 1. No rows are inserted: Command 1 starts a transaction as autocommit is off. Command “Begin” does not start another transaction as a transaction is already running (warning that a transaction is already running) Whole transaction is rolls back because of duplicate key error. 1p – correct answer, 1p – correct explanation | 2 |  |  |
| Total | **5** |  |  |

**Task 2**: Transactions

|  |  |  |  |
| --- | --- | --- | --- |
| 2.1 Expected Result | Possible Points | Your Points | Remarks |
| T2 read(A) T2 write(A) T1: read(B) T2 read(C) T3 read(B) T3 read(C) T2 write(C) T3 write(B) T1: read(B) T2 read(A) T2 write(A) T2 read(C) T3 read(B) T3 read(C) T2 write(C) T3 write(B) T1: read(B) T3 read(B) T3 read(C) T2 read(A) T2 write(A) T2 read(C) T2 write(C) T3 write(B) T1: read(B) c T3 read(B) T3 read(C) T3 write(B) c T2 read(A) T2 write(A) T2 read(C) T2 write(C) | 3 |  |  |

2.2 Strict 2pl

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **T1** | **T2** | **T3** | **Strict 2pL** | **Possible points** | **Your points** | **Remark** |
|  | Read a |  | T2 s-lock(a), read(a) | 3 |  |  |
|  | Write a |  | T2 x-lock(a), write(a) |
| Read b |  |  | T1 s-lock(b), read b  Commit and unlock b |
|  | Read c |  | T2 s-lock c, read c |
|  |  | Read b | T3 s-lock b, read b |
|  |  | Read c | T3 s-lock c, read c |
|  | Write c |  | Wait for x-lock on c, cannot write as T3 holds s lock |
|  |  | Write b | Upgrade s-lock b to x-lock b – possible because T1 unlocked b, Write b  Commit and unlock b and c |
|  | Write c |  | Gets x-lock c, write c  commit and unlock a, c |

2.3 MVCC Reapeatable Read

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **T1** | **T2** | **T3** | **MVCC repeatable read** | **Possible points** | **Your points** | **Remark** |
|  | Read a |  | T2: Reads version a0 | 3 |  |  |
|  | Write a |  | T2: Writes version a1, sets version a0 as invalid |
| Read b |  |  | T1: Reads version b0, commits |
|  | Read c |  | T1: Reads version c0 |
|  |  | Read b | T3: Reads version b0 |
|  |  | Read c | T3: Reads version c0 |
|  | Write c |  | T2: writes version c1, sets version c0 invalid, commits |
|  |  | Write b | T2: writes version b1, sets version b0 invalid, commits |

|  |  |  |  |
| --- | --- | --- | --- |
| Total | 9 |  |  |

Task 3: Waits-for-Graph

|  |  |  |  |
| --- | --- | --- | --- |
| Expected Result | Possible Points | Your  Points | Remarks |
| Transaction n+1 requesting a s-lock: Only 1 transaction can hold a x-lock on any object at any time. S-locks on objects are compatible locks. When another t requests a s-lock and gets involved in the waits-for-graph, this means that one of the existing transactions holds an x-lock on the object. 🡪 the new transaction waits for exactly this one t holding the x-lock🡪 1 more edge is added to the graph | 3 |  |  |
| Transaction n+1 requesting a x-lock: Worst case is that all n transactions involved in the waits-for-graph already hold a s-lock on the object. The new transaction (n+1) needs to wait until all n transaction release their s-locks. So, n edges would be added as maximum to the existing waits-for-graph. | 3 |  |  |
| Total | 6 |  |  |

Task 4: Write Skew

|  |  |  |  |
| --- | --- | --- | --- |
| Expected Result | Possible Points | Your  Points | Remarks |
| T1 / T2: A "predicate lock" is put on all rows that match the search result of select count(\*) from test\_ssi where on\_duty = true. 🡪 T1 updates, T2 updates T1 can commit because there is no committed write on any of the rows with predicate lock that could have affected its prior read result. T2 cannot commit because there is a committed write on one of the rows with predicate lock that could have affected its prior read result. T2 cannot know whether the write actually affected the read result. For the serialization error to happen, it is enough that the read result COULD have been affected. False positive. 🡪 t2 rolls back with a serialization error | 2 |  |  |
| Total | 2 |  |  |

Laptop Tasks:

|  |  |  |  |
| --- | --- | --- | --- |
| Expected Result | Possible Points | Your  Points | Remarks |
| Write Skew   * Postgres throws a serialization error which – in this case – is a false positive. * TransactionIDs follow each other, both transactions update * Write of 1st transaction that commits is valid and durable, 2nd transaction rolls back * Xmin, Xmax of 1st transaction update show the new valid row version after commit (Xmax = 0) * Xmin, Xmax od 2nd transaction show old Xmin and the updating t as Xmax | 8 |  |  |
| Extension of lesson DB   * Building as strong, room as weak entity with according FK / PK part referencing building (2p) * Lesson table with composite FK (building / room) to room table (3p) * Lesson table with yet another unique key: lessontime - building – room (3p) | 8 |  |  |
| De-normalized document storage:   * Query insertion time of document (1p) * Arrays: violate 1st NF – if example provided: 1.5p * Embedded documents: violate 2nd / 3rd NF: if example provided: 1.5p | 4 |  |  |
| Total | 20 |  |  |

**Total Points:**