ECE 356 Lab3 Group 10

Victor Yan
vlyan
20612514

Lun Jing
l7jing
20558988

Part One

The table below shows the data collected regarding reading through a second session at different points in time while the first session is writing, with both sessions having autocommit turned OFF. All instances of the tests occur after at least the write session has completed the UPDATE (write) query, but with no other reads or commits having occurred.

Left - Write Session Right - Read Session	**Serializable**	**Repeatable Read**	**Read Committed**	**Read Uncommitted**
Serializable	No commits: delayed until timeout: “ERROR 1205 (HY000): Lock wait timeout exceeded; try restarting transaction” After write session commit: 103 After write & read session commit: 103	No commits: 123 After write session commit: 123 After write & read session commit: 103	No commits: 123 After write session commit: 103 After write & read session commit: 103	No commits: 103 After write session commit: 103 After write & read session commit: 103
Repeatable Read	No commits: delayed until timeout: “ERROR 1205 (HY000): Lock wait timeout exceeded; try restarting transaction” After write session commit: 103 After write & read session commit: 103	No commits: 123 After write session commit: 123 After write & read session commit: 103	No commits: 123 After write session commit: 103 After write & read session commit: 103	No commits: 103 After write session commit: 103 After write & read session commit: 103

Read Committed	No commits: delayed until timeout: “ERROR 1205 (HY000): Lock wait timeout exceeded; try restarting transaction” After write session commit: 103 After write & read session commit: 103	No commits: 123 After write session commit: 123 After write & read session commit: 103	No commits: 123 After write session commit: 103 After write & read session commit: 103	No commits: 103 After write session commit: 103 After write & read session commit: 103
Read Uncommitted	No commits: delayed until timeout: “ERROR 1205 (HY000): Lock wait timeout exceeded; try restarting transaction” After write session commit: 103 After write & read session commit: 103	No commits: 123 After write session commit: 123 After write & read session commit: 103	No commits: 123 After write session commit: 103 After write & read session commit: 103	No commits: 103 After write session commit: 103 After write & read session commit: 103

In general, we found that the results of the reads only depended on the the transaction isolation level of the second (read) session and did not seem to change when the transaction isolation level of the first (write) session changed.The changes between different transaction isolation levels for the second session seemed to match up with the types of reads permitted as discussed in class (i.e. with respect to phantom reads, non-repeatable reads, and dirty reads).

When the read session had its transaction isolation level to serializable, the session was found to stall while reading after the write session wrote with an update query, and would eventually time out unless the write session committed what it had just updated. After the write session has committed, the read session would then see the updated changes from the write session through its read without delay. This behaviour correspond with disallowing phantom reads, in that the value of the read query in the second session may change due to the first session’s uncommitted updates. Thus, the second session is stalled until the first session’s transaction is fully complete before finish processing its read.

The other three isolation levels do not have this stalling behaviour even with the first session’s uncommitted update, which indicates that they all allow phantom reads to occur. The repeatable read level for the read session indicates a value of 123 (i.e. no change) when reading before the first session has committed, and also a value of 123 after the first session has committed, only updating to 103 after the second session has also committed after the first session has. This shows that repeatable read disallows non-repeatable reads, meaning that the value of the same read during one transaction must remain consistent and have the same value. Regardless of whether a write transaction is or has finished occurring during the read session's transaction, the read should not be affected.

The read committed level differs from repeatable read here in that it does allow non-repeatable reads, meaning that the value of the same read *can* change during the same transaction. This is clearly shown as the value read before the write session committed is 123, but the value after is 103.

Both repeatable read and read committed do not allow dirty reads, which are reads of values from writes that have yet been committed, which is why despite the write session having completed its write (update query), if it has not committed the transaction, then the read session will still read the old value of 123 (Note: serializable also prevents dirty reads, but its behaviour cannot be seen here as it also prevents phantom reads, and so it stalls if the write session does not commit its current transaction). The last level, read uncommitted, differs from read committed by also allowing these dirty reads, as seen from the value read immediately after the write session's write being 103, indicating that it is reading the value from a write that has yet to be committed.

When manipulating the autocommit values for the read and write sessions, the data collected indicated unsurprising result with respect to the original table. When autocommit was turned on for the write session and off for the read session, similar data was found in the table above, with the first data points removed from each cell (the data regarding "No commits") since upon writing in the write session, the transaction is immediately committed as well. The data still remained the same.

The only potential exception to this was when the read session was set to the repeatable read level. Depending on when the first read of the SELECT query occurs, the value may differ from each transaction (though it always remains the same within the same transaction). For example, if the read session read first before the write occurs in the write session, then the read session will give a value of 123, and will stay as 123 regardless of the number of write transactions completed by the write session. However, if the write session finishes one write before the read session has made any reads in its current transaction, then once the read session reads, it will take on the value it sees currently (103), and no subsequent writes will affect this value. This can be repeated with any write so long as the initial read of the read session's current transaction has yet to occur. It's understandable that the updated value can be seen in the initial read, but cannot be in consequent reads since the repeatable read level prevents non-repeatable reads. Thus, once the initial read of the value has occurred, it must remain that same value regardless of any other writes that follow.

When manipulating the autocommit values for the read session to be 1 whereas the autocommit values for the write session to be 0, the results of the read session will depend on the transaction isolation level of read session and the time write session commit. When the read session’s transaction isolation level is serializable, the result is same as what we have in the table above, read session will not have any results until write session commit; otherwise it will through the timeout error. For the other cases, the second value from each cell, which defined “after session commit”, will not be considered and other data stay unchanged. This is because when the read session is auto commit, it immediately return what is read from cell, which has been defined in the previous paragraph.

Part Two:

See 356lab3.sql for procedure code

```
-- success case
CALL switchSection(‘ECE356’, 1, 2, 1191, 10, @errorCode);
SELECT @errorCode; -- should be 0
SELECT * FROM Offering LEFT OUTER JOIN Classroom USING (roomID);
CALL switchSection(‘ECE356’, 2, 1, 1191, 10, @errorCode);

CALL switchSection(‘ECE356’, 1, 2, 1191, 15, @errorCode);
SELECT @errorCode; -- should be 0
SELECT * FROM Offering LEFT OUTER JOIN Classroom USING (roomID);
CALL switchSection(‘ECE356’, 2, 1, 1191, 15, @errorCode);

-- error case: not exists in Offering:
CALL switchSection(‘ECE999’, 1, 2, 1191, 4, @errorCode);
SELECT @errorCode; -- should be -1

CALL switchSection('ECE356', 3, 1, 1191, 4, @errorCode);
SELECT @errorCode; -- should be -1

CALL switchSection('ECE356', 2, 1, 1190, 4, @errorCode);
SELECT @errorCode; -- should be -1

-- error case: invalid quantity (non-positive value)
CALL switchSection(‘ECE356’, 1, 2, 1191, 0, @errorCode);
SELECT @errorCode; -- should be -1
SELECT * FROM Offering LEFT OUTER JOIN Classroom USING (roomID);

CALL switchSection(‘ECE356’, 1, 2, 1191, -5, @errorCode);
SELECT @errorCode; -- should be -1
SELECT * FROM Offering LEFT OUTER JOIN Classroom USING (roomID);

-- error case: same section
CALL switchSection(‘ECE356’, 1, 1, 1191, 10, @errorCode);
SELECT @errorCode; -- should be -1
SELECT * FROM Offering LEFT OUTER JOIN Classroom USING (roomID);

-- error case: exceeding capacity
CALL switchSection(‘ECE356’, 1, 2, 1191, 20, @errorCode);
SELECT @errorCode; -- should be -3
SELECT * FROM Offering LEFT OUTER JOIN Classroom USING (roomID);
```

```
UPDATE Offering SET enrollment = 10
WHERE courseID = ‘ECE356’ AND section = 1 AND termCode = 1191;

CALL switchSection(‘ECE356’, 1, 2, 1191, 10, @errorCode);
SELECT @errorCode; -- should be 0
SELECT * FROM Offering LEFT OUTER JOIN Classroom USING (roomID);
CALL switchSection(‘ECE356’, 2, 1, 1191, 10, @errorCode);

-- error case: lacking capacity to switch
CALL switchSection(‘ECE356’, 1, 2, 1191, 20, @errorCode);
SELECT @errorCode; -- should be -2
SELECT * FROM Offering LEFT OUTER JOIN Classroom USING (roomID);
```

Part Three

	Without indexing Run_time (µs)	With Indexing Run_time (µs)
1	134199684.0000	950093.0000
2	130487853.0000	753984.0000
3	151620080.0000	731960.0000
4	129159487.0000	631194.0000
5	129370706.0000	668656.0000

```
For Performance without indexing
MAX: 151620080.0000 µs
MIN: 129159487.0000 µs
AVG: 134967562      µs
```

```
For Performance with indexing
MAX: 950093.0000 µs
MIN: 631194.0000 µs
AVG: 747177.4    µs
```

```
ALTER TABLE Master ADD PRIMARY KEY (playerID);
ALTER TABLE Teams ADD PRIMARY KEY (yearID, teamID);
ALTER TABLE Batting
ADD PRIMARY KEY (playerID, yearID, teamID);
ALTER TABLE Batting
ADD FOREIGN KEY (playerID) REFERENCES Master(playerID);

CREATE INDEX Batting_RBI on Batting (RBI) Using BTREE;
CREATE INDEX Batting_HR on Batting (HR) Using HASH;
```

As we can see, adding indexing will improve the performance dramatically. After checking the events transactions histories of the repeated query, there is, on average, a speed improvement by a factor of 180.6 by adding the appropriate indexes.

To analyze how the primary keys and foreign keys can help to improve the performance, we removed all the primary keys and foreign keys, in that case, we have only a B-tree indexing for RBI and a Hash indexing for HR. The result is really surprising, the performance is actually getting worse. It takes about 20 minutes but still have no responses.

In this case, we tried an alternative method, we only add indexing on playerID, but do not including anything on HR or RBI. The performance is about the same as the optimized results in the table above.

It concludes that the major factor that optimized the performance of the query is the indexing of playerID; it can be created by either adding the playerID as foreign key or adding a specific indexing for it.