Database Tuning – Assignment 6

Concurrency Tuning

A2

Baumgartner Dominik, 0920177

Dafir Thomas Samy, 1331483

Schörgnhofer Kevin, 1421082

January 10, 2017

Setup

All experiments were executed on the computers in the RÜR. JDBC was used to send transactions to the database server (biber). Time was measured using the *time* shell command.

Task 1

Read Committed

Throughput and correctness for solution (a) with serialization level `READ COMMITTED`.

#Concurrent Transactions	Throughput [transactions/sec]	Correctness
1	18.9	100%
2	32.6	52%
3	45.4	70%
4	56.3	59%
5	64.1	55%

Serializable

Throughput and correctness for solution (a) with serialization level `SERIALIZABLE`.

#Concurrent Transactions	Throughput [transactions/sec]	Correctness
1	19.23 - 20.41	100 %
2	20.41 - 22.22	100 %
3	19.23 - 21.74	100 %
4	20.0 - 28.57	100 %
5	20.83 - 34.99	100 %

Task 2

Read Committed

Throughput and correctness for solution (b) with serialization level `READ COMMITTED`.

#Concurrent Transactions	Throughput [transactions/sec]	Correctness
1	23.5	100%
2	39.1	100%
3	52.6	100%
4	58.9	100%
5	67.6	100%

Serializable

Throughput and correctness for solution (b) with serialization level `SERIALIZABLE`.

#Concurrent Transactions	Throughput [transactions/sec]	Correctness
1	21.74 - 23.8	100 %
2	26.31 - 40.0	100 %
3	27.78 - 40.0	100 %
4	35.71 - 45.45	100 %
5	37.04 - 43.47	100 %

Task 3: Discussion

Task 1

As expected using isolation level read committed led to a much higher throughput with increasing number of concurrent transactions. In the case of serializable the gain was not as dramatic. This is due to the conflicts caused by the serializable isolation level leading to transactions being executed several times before committing which leads to lower throughput.

Serializable uses strict 2-phase locking for read and write operations: only one transaction can read/write data at a time. Locks are released after a transaction commits. Transactions trying to modify the same dataset as the current transaction have to wait until it commits. Any changes done by other transactions are rolled back.

Error rates are also as expected: Using serializable leads to no errors at all since it yields the same results as a serial execution of all transactions. In the case of read committed non repeatable reads occur since every transaction sees the database as it was at the time the transaction started. E.g. if a transaction manipulates *balance* by subtracting 1 it does not notice that other transactions might have done the same in the meantime. When this transaction commits it writes the old value decreased by 1 in to the db. This leads to calculation errors but increases the throughput since no transactions are rolled back and re-executed.

Task 2

For both isolation levels we get 100% correctness, as expected.
For the isolation level serializable it is obvious that the concurrent transactions have no

effects to each other (that is kind of the definition of this isolation level), which results in 100% correctness.
To understand the 100% correctness of the read committed isolation level, we have to take a look at the transaction: it consits of two update statements, which are both enclosed and do not relate on a value from a previous select statement (or similar). So there is no way an update statement from another transaction could interfere with the current update statement.
The throughput increases as the number of concurrent transactions increases (as expected).
Compared to serializable the throughput of read committed is higher, because a concurrent transaction B which starts after transaction A, waits until A commits and commits then itself. With serializable, transaction B aborts after A commits. This leads to way more queries needed for the isolation level serializable, hence to a lower throughput.

Update

Transaction:
Serializable: Updates are only efective if the whole transaction commits. Otherwise all changes are rolled back. This also explaines the result of our concurrent serializable experiment: Each Transaction accesses a different tuple and adds 1, but all transactions access the same account (100) to subtract one. Even though the first update could be executed without conflict the whole transaction is aborted due to a conflict caused by the second update. This prevents the transaction from leaving the databse in an inconsistent state (1 added to tuple but not subtracted from other tuple).

Read committed: Updates have to wait until the other concurrent transaction owning the lock commits. The locks can then be aquired and the transaction can update and commit. This explains the errors in query 1. Although an error (concurrent access on same tuple) is detected the transaction waits and is then allowed to commit its result even though it was calculated using an outdated value. In query 2 no values are read from the database. the transaction just adds 1 to the tuple its responsible for and subtracts 1 from the shared tuple (account = 0). No transaction is dependent on results another transaction has calculated. Not even interrupting the transaction between the two update queries leads to errors since it does not matter when the calculation is executed. After all are finished 100 has been subtracted from the 0 account and 1 added to all others.

Time in hours per person: **6**

Information Sources in general:
Lecture Notes
`https://www.postgresql.org/docs/9.6/static/transaction-iso.html`